

Ruhr-Universität Bochum  
Fakultät für Elektrotechnik und Informationstechnik  
Veranstaltung: Embedded Multimedia  
Sommersemester 2015  
Dozent: Dr. Wolfgang Theimer

## **PräsiBert – eine Blackberry-Applikation für Präsentationen und Vorträge**

### **Abschlussbericht**

Eingereicht von:

Arno Däuper  
Kronenstr. 67  
44789 Bochum  
Arno.Daeuper@rub.de  
108 0131 16375

Niklas Kröger  
Deismannstr. 52  
44795 Bochum  
Niklas.Kröger@rub.de  
108 0102 51515

Inga Quatuor  
Poststr. 141  
44809 Bochum  
Inga.Quatuor@rub.de  
108 0052 28213

Sebastian Schwanewilms  
Oskar-Hoffmann-Str. 104  
44789 Bochum  
Sebastian.Schwanewilms@rub.de  
108 0132 75897

Jan Zimmer  
Wittener Str. 76  
44789 Bochum  
Jan.Zimmer@rub.de  
108 0112 22492

# Inhalt

1	Einleitung	2
2	Das Projekt	3
2.1	Beschreibung	3
2.2	Softwarearchitektur	6
2.3	Use Cases	7
3	Projektorganisation	13
3.1	Spezifikationsphase	13
3.2	Implementierungsphase	14
3.3	Zusammenführung der Komponenten	15
4	Die Komponenten	16
4.1	Server	17
4.2	Client	25
4.3	Netzwerkschicht	48
4.4	Kommunikation/Nachrichtenformat	54
4.5	GUI	60
4.6	Audio	66
4.7	Bildschirmausgabe über HDMI	69
4.8	Gestensteuerung	70
5	Systemintegration	78
6	Projektreview	82
7	Fazit	83
8	Anhang	84

# 1 Einleitung<sup>1</sup>

Aufgabe und Bestandteil der Lehrveranstaltung „Embedded Multimedia“ an der Ruhr-Universität Bochum ist die Entwicklung einer Applikation für das Blackberry Z10. Das Standardprojekt dieser Lehrveranstaltung ist die Entwicklung einer Präsentations-Applikation. Diese soll eine Präsentation von dem Blackberry Z10 aus steuern und über das WLAN-Netz an andere Blackberry Geräte verteilen können.

Die Themengebiete, welche für die Entwicklung der Applikation notwendig sind, werden in der Vorlesung der Veranstaltung gezeigt und erklärt. Unter anderem befasst sich die Vorlesung mit den Bereichen Betriebssysteme, Netzwerkkommunikation, Signalverarbeitung, Audiosignale, Gestensteuerung, grafische Benutzeroberflächen, Nebenläufigkeit und geteilte Anwendungen. Den Studierenden soll so das nötige Grundlagenwissen für die eigenständige Entwicklung der Applikation vermittelt werden.

Dieser Bericht dokumentiert die Entwicklung der Blackberry Applikation *PräsiBert* von *Team 2*. Wir, die Mitglieder dieses Teams, sind: *Arno Däuper, Niklas Kröger, Sebastian Schwanewilms, Inga Quatuor* und *Jan Zimmer*.

Zu Beginn der Lehrveranstaltung entschieden wir uns für die Entwicklung des Standardprojekts mit zusätzlichen, selbst gesetzten Erweiterungen für die Applikation. Die Details hierzu werden im zweiten Kapitel erläutert. Im dritten Kapitel wird Organisation innerhalb des Teams beschrieben. Das vierte Kapitel befasst sich mit den verschiedenen Komponenten der Applikation und erklärt, wie diese im Einzelnen funktionieren. Die Quellcode-Dokumentation im Anhang, welche mit Doxygen angefertigt wurde, gibt weitere Aufschlüsse über die Funktionsweise der Applikation. Im fünften Kapitel beschreiben wir die Zusammenführung aller Komponenten zu einem Gesamtsystem. Dies soll zeigen, wie unsere unabhängig voneinander entwickelten Komponenten problemlos zu einer Gesamtapplikation zusammengeführt werden konnten. Der Bericht endet mit einem Projektreview und einem Fazit zum gesamten Projekt.

---

<sup>1</sup> Autor: Niklas Kröger

## 2 Das Projekt<sup>2</sup>

In diesem Kapitel erfolgt zunächst eine umfassende Beschreibung des Projekts *PräsiBert*. Hierfür werden das geplante Einsatzumfeld und der gesamte Funktionsumfang aus Anwendersicht erläutert. Auf Grundlage dieser Anforderungen wurde eine Softwarearchitektur entworfen, die im zweiten Abschnitt präsentiert wird. Abschließend werden im dritten Abschnitt die *Use Cases* vorgestellt, die wir im Laufe der Spezifikation entworfen haben und die einen detaillierten Einblick in die Funktionsweise der Applikation erlauben.

### 2.1 Beschreibung

Als Ausgangspunkt haben wir das Standardprojekt für die Veranstaltung „Embedded Multimedia“ gewählt, einen „Mobile Presenter with A/V Remote Control“ für das Blackberry Z10. Dieser „Mobile Presenter“ soll es einem Anwender erlauben, auf seinem Blackberry Präsentationsdaten (Bilder oder optional Videos) auszuwählen, die dann auf einem angeschlossenen Beamer oder Monitor gezeigt werden. Zur Steuerung der Präsentation sollen audiovisuelle Signale – z. B. Händeklatschen – verwendet werden. Außerdem soll die Präsentation an weitere Blackberrys via WiFi verteilt werden.

Wir haben uns überlegt, dass es sinnvoll wäre, wenn unsere Applikation nicht nur für Einzelvorträge einzusetzen ist, sondern auch für mehrere konsekutive Vorträge mit wechselnden Vortragenden, wie es zum Beispiel auf Konferenzen häufig der Fall ist. Eine Konsequenz hieraus ist, dass es sich für unser Projekt bei Organisator und Vortragendem nicht zwingend um eine Person handelt. Für die Entwicklung der Applikation *PräsiBert* gibt es also drei zu berücksichtigende Anwendergruppen: die Organisatoren, die Vortragenden und das Publikum. Diese *Stakeholder* stellen unterschiedliche Anforderungen an das Produkt und wir haben versucht, möglichst viele dieser Anforderungen durch Anpassungen des Standardprojekts abzudecken.

Eine Anforderung, die alle Anwender erwartungsgemäß teilen, ist, dass *PräsiBert* einfach und komfortabel zu bedienen ist. Entsprechend soll unsere ganze Anwendung

---

<sup>2</sup> Autor: Inga Quatuor

diesem Grundsatz folgen und die grafische Benutzeroberfläche möglichst schlicht und selbsterklärend gestaltet sein.

Aus organisatorischer Sicht sollte das Setup für die Vorträge nur einmal erfolgen müssen – zu Beginn der Veranstaltung – und nicht für jeden Vortrag. Analog hierzu möchte ich mich aus Zuhörersicht nicht für jeden Vortrag neu anmelden müssen. Hier drängt sich bereits eine Server-Lösung auf, die wir auch umgesetzt haben und die im nächsten Abschnitt „Softwarearchitektur“ näher erläutert wird.

Für den Organisator einer Konferenz oder vergleichbaren Veranstaltung dürfte eine Archivierung der Vorträge von großem Interesse sein, beispielsweise um die einzelnen Vorträge später online zur Verfügung zu stellen. Dafür müssen zum einen die Präsentationsdaten – bei *PräsiBert* nur Bilder und keine Videos – gespeichert werden und zum anderen natürlich der Vortrag selbst. Hierzu wird über die gesamte Dauer eines Vortrags ein Audiomitschnitt angefertigt.

Als gemeinsame Anforderung des Organisators und des Vortragenden ist zudem eine gewisse Sicherheit der Applikation zu gewährleisten. Es darf beispielsweise nicht passieren, dass die Steuerung der Präsentation unbefugt „gekapert“ wird und die Folien von einem anderen Device als dem des Vortragenden aus gewechselt werden.

Aus Sicht des Vortragenden ist es von größter Wichtigkeit, dass *PräsiBert* ihn bei seinem Vortrag unterstützt und während des Vortrags nur wenig Aufmerksamkeit erfordert. Notwendige Eingaben – z. B. das Auswählen der Präsentationsdateien und das Aktivieren oder Deaktivieren bestimmter Optionen – sollen also möglichst vor dem eigentlichen Vortrag erfolgen können. Funktionen, die unabhängig von Eingaben des Vortragenden erfolgen können, sollten automatisiert ausgeführt werden. Beispielsweise ist es nicht notwendig, dass sich der Vortragende mit der Verbindung von Zuhörer-Devices auseinandersetzt oder explizit die Audioaufnahme bei Vortragsbeginn startet.

Während des Vortrags steht die unkomplizierte Steuerung der Präsentation im Vordergrund. Hierzu gibt es die klassische Möglichkeit, die Folien über Buttons in der grafischen Benutzeroberfläche vor- und zurückzuschalten. Alternativ kann der Vortragende aber auch das Feature „Gestensteuerung“ benutzen. Hierzu legt er sein Blackberry am besten stationär ab, so dass es einen konstanten hellen Hintergrund gibt,

und bewegt dann die Hand über den Sucher der Frontkamera. Eine Bewegung von links nach rechts schaltet eine Folie nach vorne und eine Bewegung in die umgekehrte Richtung schaltet eine Folie zurück. Außerdem soll es eine Möglichkeit geben, über das GUI direkt zu einer bestimmten Folie zu springen, um beispielsweise besser auf Fragen aus dem Publikum reagieren zu können.

Die nächste Erweiterung, die wir vorgenommen haben, betrifft genau solche Fragen. Gerade bei größeren Räumlichkeiten sind Wortmeldungen aus dem Publikum häufig leicht zu übersehen und bei mehreren Wortmeldungen ist es meistens schwer zu sagen, wer zuerst dran ist. *PräsiBert* soll den Vortragenden hier unterstützen und Publikumsfragen verwalten. Wenn ein Zuhörer eine Frage hat, kann er mit seinem Blackberry eine Redeanfrage absenden. Der Vortragende bekommt diese Anfrage auf seinem GUI angezeigt und kann diese annehmen oder ablehnen. Bei mehreren Anfragen wird automatisch eine Warteschlange erzeugt. Wird einem Zuhörer das Wort erteilt, soll auf seinem Device ein Audiomitschnitt starten, so dass die bereits angesprochene Archivierung der Vorträge vollständig ist.

Die Frage der Räumlichkeit betrifft auch die nächste Anpassung. So ist es nicht garantiert, dass die Position des Vortragenden gleichzeitig der beste Standort ist, um ein Ausgabemedium wie einen Beamer anzuschließen. Von daher wäre es vorteilhaft, wenn jedes im Netzwerk angemeldete Blackberry, also auch das eines Zuhörers, als *Output Device* dienen kann. Konsequenterweise, gibt es keinen Grund einzuschränken, wie viele Ausgabemedien angeschlossen werden dürfen. Folglich ist es mit *PräsiBert* möglich, mehrere Beamer und / oder Monitore gleichzeitig anzusprechen, sofern diese Ausgabegeräte per HDMI an ein Blackberry angeschlossen sind.

Abschließend gibt es einige Ideen, die wir als lohnende Erweiterungen des Projekts diskutiert haben, aber aufgrund mangelnder Ressourcen nicht umgesetzt haben. Hierzu gehört insbesondere das *Post Processing* der gesammelten Vortragsdaten, also der Präsentationsbilder und der Audiomitschnitte. Die Grundlagen dafür stellt *PräsiBert* jedoch bereit, da sämtliche Daten mit präzisen Zeitstempeln versehen werden. Eine weitere sinnvolle Ergänzung bestünde darin, vielfältige Präsentationsformate (diverse Bildformate, Powerpoint, PDF) zu unterstützen. Außerdem könnte der IT-Sicherheitsaspekt ausgebaut werden, indem nicht nur die Kommunikation des

Vortragenden authentifiziert erfolgt, sondern die Kommunikation aller Vortragsteilnehmer.

## **2.2 Softwarearchitektur**

Wie im vorigen Abschnitt schon angesprochen wurde, haben wir uns entschieden, eine Serverlösung einzusetzen, um die Basis für die verteilte Anwendung vom Device des Vortragenden zu entkoppeln. Das hat den Vorteil, dass der Server nur zu Beginn der Veranstaltung eingerichtet werden muss und ab dann – auch über mehrere Vorträge hinweg – die Vortragsteilnehmer verwalten kann, die Kommunikation zwischen den Teilnehmern weiterleitet und zudem die Archivierung der Vortragsdaten übernehmen kann.

Für die Client-Seite war schnell deutlich, dass wir am besten zwei eigenständige Applikationen entwickeln. Eine MasterClient-Anwendung, die vom jeweiligen Vortragenden eingesetzt wird, und eine ListenerClient-Anwendung, die von allen Zuhörern benutzt wird. Diese drei Applikationen – Server, MasterClient und ListenerClient – können dabei theoretisch auf ein und demselben Blackberry laufen.

Der Server ist dafür zuständig, sämtliche Clients, die sich bei ihm einloggen, zu verwalten. Dabei muss er beispielsweise sicherstellen, dass sich nur ein MasterClient gleichzeitig anmelden kann. Außerdem läuft sämtliche Kommunikation über den Server. Möchte zum Beispiel ein ListenerClient eine Frage stellen, wird dieser Request an den Server geschickt und der Server leitet ihn an den MasterClient weiter. Das hat den großen Vorteil, dass sich die Clients untereinander nicht kennen müssen. Zu guter Letzt verwaltet der Server alle Daten, das heißt, er bekommt die Präsentation vom MasterClient geschickt und übernimmt deren Verteilung an alle angemeldeten ListenerClients. Alle Audiomitschnitte, die während des Vortrags angefertigt werden, werden ebenfalls an den Server geschickt und dort gespeichert.

Der MasterClient bietet einen größeren Funktionsumfang als der ListenerClient, da der Vortragende natürlich mehr Einfluss auf den Vortrag nehmen können muss als die Zuhörer. Neben der Auswahl der Präsentation und der Steuerung derselben kann der MasterClient auch eine Reihe von Einstellungen wie das Aktivieren oder Deaktivieren

der Gestensteuerung vornehmen. Im Hintergrund läuft zudem der *AudioRecorder*, sobald der Vortrag gestartet wurde.

Die Hauptaufgabe des *ListenerClient* ist es, die aktuelle Folie der Präsentation, die er vom Server mitgeteilt bekommt, anzuzeigen. Ergänzend hierzu hat er die Möglichkeit, eine Redeanfrage abzuschicken und falls diese bewilligt wird, startet auch beim *ListenerClient* eine Audioaufnahme.

Ausgehend von dieser Struktur haben wir uns entschieden, die weitere Softwarearchitektur ebenfalls möglichst modular zu gestalten. Das hat folgende drei Vorteile: Erstens gibt es mehrere Komponenten, die von mehr als einer Applikation benutzt werden, beispielsweise den *AudioRecorder* um Audiomitschnitte anzufertigen oder die Netzwerkschicht, um Daten zwischen den Clients und dem Server zu übertragen. Zweitens erlaubt die modulare Struktur eine sinnvolle Arbeitsteilung innerhalb des Teams. Drittens garantiert diese Struktur eine saubere funktionale Entkopplung auf Implementierungsebene und erleichtert die Erweiterbarkeit sowie Portierbarkeit<sup>3</sup> der Software.

Die einzelnen Komponenten – Server, Clients, Netzwerkschicht, Audiorekorder, GUI und Gestensteuerung – werden im vierten Kapitel detailliert beschrieben.

## 2.3 Use Cases

Im Folgenden werden die *Use Cases* in tabellarischer Form vorgestellt, die wir im Rahmen der Spezifikation erstellt haben, um die Funktionsweise der Applikation möglichst genau zu beschreiben. Die *Use Cases* sind hierbei aus Anwendersicht – Organisator, Vortragender oder Zuhörer – beschrieben. Die *Use Cases* sind soweit möglich in der Reihenfolge aufgeführt, in der sie auch bei einem Einsatz der Präsentationssoftware auftreten.

Einige *Use Cases* sind erst während der Implementierungsphase hinzugekommen (z. B. „Gestensteuerung aktivieren oder deaktivieren“), während an anderen *Use Cases* noch Anpassungen vorgenommen wurden. Beispielsweise wurde „Auf Redeanfrage

---

<sup>3</sup> Das Thema Portierbarkeit ist insbesondere für die Server-Applikation interessant, da es in der Praxis wünschenswert erscheint, diese Anwendung eher auf einem PC als auf einem Smartphone laufen zu lassen. Durch die modulare Architektur von PräsiBert ist eine solche Portierung mit relativ geringem Aufwand realisierbar.



reagieren“ dahingehend verändert, dass bei Ablehnung alle Redeanfragen in der Warteschlange gesammelt abgelehnt werden. Auf *Quantifications* wurde verzichtet, da wir für die einzelnen *Use Cases* keine sinnvollen messbaren Anforderungen formulieren konnten.

Use Case	Server starten
Scenario	Der Anwender möchte PräsiBert benutzen.
Actor	Organisator oder Vortragender
Pre-Condition	keine
User Interaction Task Sequence	<ul style="list-style-type: none"> <li>• die Server-Anwendung starten</li> <li>• IP-Adresse des Servers notieren</li> </ul>
Post-Condition	Die Server-Anwendung läuft und IP-Adresse des Servers ist bekannt.

Use Case	Vortrag vorbereiten
Scenario	Der Anwender möchte einen Vortrag halten und die notwendigen Vorbereitungen dafür vornehmen.
Actor	Vortragender
Pre-Condition	Die Server-Anwendung läuft und IP-Adresse des Servers ist bekannt.
User Interaction Task Sequence	<ul style="list-style-type: none"> <li>• die PräsiBert-Anwendung als MasterClient starten</li> <li>• beim Server anmelden (mit Authentifizierung)</li> <li>• eine lokal gespeicherte Präsentation auswählen (wird dann an den Server gesendet)</li> </ul>
Post-Condition	Der Anwender ist beim Server als MasterClient registriert. Der Server verfügt über die Präsentationsdatei. Der Anwender kann nun den Vortrag starten.

Use Case	Vortrag zuhören
Scenario	Der Anwender möchte einem Vortrag zuhören.
Actor	Zuhörer

Pre-Condition	Die Server-Anwendung läuft und IP-Adresse des Servers ist bekannt.
User Interaction Task Sequence	<ul style="list-style-type: none"> <li>• die PräsiBert-Anwendung als ListenerClient starten</li> <li>• beim Server anmelden (ohne Authentifizierung)</li> </ul>
Post-Condition	Der Anwender ist beim Server als ListenerClient registriert. Wurde bereits eine Präsentation an den Server übermittelt, bekommt der Anwender diese Daten auf sein Device übertragen. Wenn ein Vortrag läuft, kann er Fragen stellen (siehe Use Case „Frage stellen“).

Use Case	Ausgabemedium einrichten
Scenario	Der Anwender möchte, dass die Präsentation über sein Device an ein Ausgabemedium (z. B. Beamer) weitergeleitet wird.
Actor	beliebig
Pre-Condition	Eine Client-Anwendung (Master oder Listener) wurde gestartet.
User Interaction Task Sequence	<ul style="list-style-type: none"> <li>• das Ausgabemedium per HDMI an das Device anschließen</li> <li>• in der Client-Anwendung im Einstellungsmenü die Option „HDMI Ausgabe“ aktivieren</li> </ul>
Post-Condition	Die aktuelle Folie der Präsentation wird an das Ausgabemedium übertragen.

Use Case	Vortrag starten
Scenario	Der Anwender möchte einen Vortrag mit Präsentation starten
Actor	Vortragender
Pre-Condition	Use Case „Vortrag vorbereiten“ ist abgeschlossen: Der Anwender ist beim Server als MasterClient registriert und der Server verfügt über die Präsentationsdatei.

User Interaction Task Sequence	<ul style="list-style-type: none"> <li>• die Präsentation per Button starten</li> </ul>
Post-Condition	<p>Die erste Folie der Präsentation wird auf allen angemeldeten Devices (und Beamern etc.) angezeigt. Eine Audioaufnahme wurde beim MasterClient gestartet. Der Anwender kann die Präsentation steuern.</p>

<b>Use Case</b>	<b>Gestensteuerung aktivieren oder deaktivieren</b>
Scenario	Der Anwender möchte die Gestensteuerung aktivieren oder deaktivieren
Actor	Vortragender
Pre-Condition	Der Anwender fungiert als MasterClient.
User Interaction Task Sequence	<ul style="list-style-type: none"> <li>• um die Gestensteuerung zu aktivieren, im Optionsmenü einen entsprechenden Haken setzen</li> <li>• um die Gestenstuerung zu deaktiviert, diesen Haken entfernen</li> </ul>
Post-Condition	Das Feature Gestensteuerung ist nun aktiviert oder deaktiviert.

<b>Use Case</b>	<b>Vortrag steuern</b>
Scenario	Der Anwender möchte die Folie wechseln
Actor	Vortragender
Pre-Condition	Ein Vortrag wurde gestartet und der Anwender fungiert als MasterClient.
User Interaction Task Sequence	<p>per Gestensteuerung:</p> <ul style="list-style-type: none"> <li>• die Hand von links nach rechts über die Frontkamera des Blackberry bewegen, um eine Folie vor zu schalten bzw. von rechts nach links, um eine Folie zurück zu schalten</li> </ul> <p>per GUI:</p> <ul style="list-style-type: none"> <li>• um eine Folie vor oder zurück zu schalten, auf die entsprechenden Buttons drücken</li> </ul>

	<ul style="list-style-type: none"> <li>um direkt zu einer bestimmten Folie zu wechseln, die Foliennummer eingeben und bestätigen</li> </ul>
Post-Condition	Der Server hat den Befehl, die neue Folie anzuzeigen, an alle Clienten gesendet.

<b>Use Case</b>	<b>Redeanfrage senden</b>
Scenario	Der Anwender möchte eine Frage stellen
Actor	Zuhörer
Pre-Condition	Ein Vortrag wurde gestartet und der Anwender fungiert als ListenerClient.
User Interaction Task Sequence	<ul style="list-style-type: none"> <li>auf den entsprechenden Button drücken</li> </ul>
Post-Condition	Der Vortragende (MasterClient) bekommt den Redewunsch des Zuhörers auf seinem GUI mitgeteilt

<b>Use Case</b>	<b>Auf Redeanfrage reagieren</b>
Scenario	Mindestens ein Zuhörer hat eine Redeanfrage gesendet und der Anwender ist per GUI darüber informiert worden. Der Anwender möchte entscheiden, ob der Zuhörer seine Frage stellen darf oder nicht.
Actor	Vortragender
Pre-Condition	Ein ListenerClient hat eine Redeanfrage gestellt (siehe Use Case „Redeanfrage senden“). Haben mehrere ListenerClients Redeanfragen gesendet, wurde eine Warteschlange erzeugt.
User Interaction Task Sequence	<ul style="list-style-type: none"> <li>zur Annahme der Redeanfrage (bzw. der ersten Redeanfrage in der Warteschlange) auf den entsprechenden Button drücken</li> <li>zur Ablehnung der Redeanfrage (bzw. aller Redeanfragen in der Warteschlange) auf den entsprechenden Button drücken</li> </ul>
Post-Condition	Bei Annahme wurde dem Zuhörer, der die Anfrage gestellt hatte, dies mitgeteilt (siehe Use Case „Frage

	stellen“). Bei Ablehnung wurde dies allen Zuhörern, die Anfragen gestellt hatten, mitgeteilt.
--	-----------------------------------------------------------------------------------------------

Use Case	Frage stellen
Scenario	Der Anwender möchte eine Frage stellen
Actor	Zuhörer
Pre-Condition	Der Anwender (ListenerClient) hat zuvor eine Redeanfrage gesendet, die bewilligt wurde.
User Interaction Task Sequence	<ul style="list-style-type: none"> <li>• wenn die Frage nicht mehr aktuell ist, Redeanfrage per Button abbrechen</li> <li>• sonst fortfahren (per Button) und mündlich die Frage stellen</li> </ul>
Post-Condition	Bei Abbruch: unverändert Sonst: Die Audioaufnahme bei diesem ListenerClient wurde gestartet.

Use Case	Frage abbrechen
Scenario	Ein Zuhörer redet und der Anwender möchte dessen Redezeit beenden.
Actor	Vortragender
Pre-Condition	Ein ListenerClient hat eine Redeanfrage gestellt, die bewilligt wurde und stellt nun seine Frage (siehe Use Case „Frage stellen“)
User Interaction Task Sequence	<ul style="list-style-type: none"> <li>• auf den entsprechenden Button drücken</li> </ul>
Post-Condition	Dem Zuhörer wurde mitgeteilt, dass seine Redezeit beendet wurde. Die Audioaufnahme beim ListenerClient wurde gestoppt und die Audiodatei an den Server geschickt.

Use Case	Keine-Fragen-Modus ein- oder ausschalten
Scenario	Der Anwender möchte entweder keine weiteren Redeanfragen von Zuhörern mehr gestellt bekommen

	oder er hatte den Keine-Fragen-Modus bereits aktiviert und möchte diesen nun wieder ausschalten, um Redeanfragen erneut zuzulassen.
Actor	Vortragender
Pre-Condition	Der Anwender fungiert als MasterClient.
User Interaction Task Sequence	<ul style="list-style-type: none"> <li>auf den entsprechenden Button (Toggle) drücken</li> </ul>
Post-Condition	<p>Wenn aktiviert: Alle Redeanfragen von Zuhörern werden jetzt automatisch abgelehnt.</p> <p>Wenn deaktiviert: Redeanfragen werden an den MasterClient weitergeleitet.</p>

<b>Use Case</b>	<b>Vortrag beenden</b>
Scenario	Der Anwender möchte den Vortrag beenden.
Actor	Vortragender
Pre-Condition	Ein Vortrag wurde gestartet und der Anwender fungiert als MasterClient.
User Interaction Task Sequence	<ul style="list-style-type: none"> <li>auf den entsprechenden Button drücken</li> </ul>
Post-Condition	Die Audioaufnahme des MasterClient wurde an den Server geschickt. Der Server hat alle Dateien des Vortrags final abgespeichert. Der Server hat den MasterClient abgemeldet.

### 3 Projektorganisation<sup>4</sup>

#### 3.1 Spezifikationsphase

Das Projekt begann mit einer intensiven Spezifikationsphase, in der der Umfang der Software definiert und festgehalten wurde. Nachdem die Spezifikation der Software vollständig war, musste geplant werden, aus welchen (Software-)Komponenten / Modulen die Applikationen bestehen sollen und wie diese an die Entwickler verteilt werden.

---

<sup>4</sup> Autor: Arno Däuper

Die Aufteilung geschah nicht aus Sicht der verschiedenen Applikationen (Server, MasterClient, ListenerClient), sondern nach technischem Aspekt, wie beispielsweise Netzwerk-Kommunikation, Bildausgabe, Gestenerkennung und Sprachaufnahme, da ein Teil der Komponenten in mehr als einer Applikation genutzt wird. Somit war es z. B. nicht notwendig, den Audiorecorder für den Vortragenden und die Zuhörer getrennt zu entwickeln. Daraus entstand jedoch gleichzeitig die Notwendigkeit, mehr einheitliche Schnittstellen zu definieren.

Die Aufteilung geschah wie folgt:

- |                                    |                        |
|------------------------------------|------------------------|
| • Server:                          | Sebastian Schwanewilms |
| • Client:                          | Jan Zimmer             |
| • Netzwerk:                        | Niklas Kröger          |
| • Kommunikation/Nachrichtenformat  | Jan Zimmer             |
| • Audio:                           | Arno Däuper            |
| • GUI:                             | Jan Zimmer             |
| • Gestensteuerung:                 | Inga Quatuor           |
| • HDMI-Ausgabe:                    | Arno Däuper            |
| • Zusammenführung der Komponenten: | Sebastian Schwanewilms |

Zur Versionsverwaltung wurde die Software Git mit *github* als Server eingesetzt, welches die Möglichkeit bietet, für die verschiedenen Entwickler so genannte *branches* zu erzeugen, die am Ende mit dem *merge*-Befehl zusammengefügt wurden. Auch sind Änderungen jederzeit nachzuvollziehen und lassen sich gegebenenfalls rückgängig machen.

### 3.2 Implementierungsphase

Mithilfe der gewonnenen Spezifikation war es jedem Entwickler nun möglich, mit der individuellen Implementierung der Komponenten zu beginnen. Die Planung sah vor, möglichst frühzeitig lauffähige Netzwerk-Komponenten zu haben, da dies eine wichtige Basis der gesamten Applikation darstellt.

Die Gruppe traf sich in dieser Phase regelmäßig, um sich gegenseitig über den Stand der Entwicklung zu informieren und reichlich Erfahrungen auszutauschen, was

insbesondere den Umgang mit der weitgehend unbekannten Entwicklungsumgebung und den Libraries betraf.

### **3.3 Zusammenführung der Komponenten**

Die Komponenten wurden schrittweise zusammengeführt, da nicht alle Komponenten gleichzeitig verfügbar waren und ein umfangreiches Debugging notwendig war.



## 4 Die Komponenten<sup>5</sup>

In diesem Kapitel werden die einzelnen Komponenten von *PräsiBert* beschrieben. Dafür werden für jede Komponente zunächst die gewünschte Funktionalität und die daraus resultierenden Anforderungen formuliert. Danach wird die Struktur der Implementierung mit Hilfe von Diagrammen erläutert. Abschließend wird auf Details eingegangen wie bestimmte Implementierungsentscheidungen, die genauere Betrachtung verdienen, oder aber auch Probleme, die während der Entwicklung aufgetreten sind.

---

<sup>5</sup> Autoren siehe einzelne Abschnitte

## 4.1 Server<sup>6</sup>

Der Server stellt die zentrale Komponente der Anwendung dar. Ohne ihn ist eine Präsentation mittels PräsiBert nicht möglich.

### **Funktionalität**

Die Funktionen des Servers lassen sich hauptsächlich in vier Kategorien einordnen:

- Clientverwaltung
- Präsentationsverwaltung
- Datenhaltung
- Kommandoweiterleitung

Die Kommunikation mit dem Server läuft über ein TCP/IP-Netzwerk ab. Die Implementierung der Netzwerkfunktionalität ist im Unterabschnitt „Netzwerkschicht“ genauer erläutert. Zur Kommunikation zwischen Server und Clients werden Nachrichten im XML-Format verwendet (siehe Abschnitt 4.4).

### ***Clientverwaltung***

Der Server ist in der Lage einen MasterClients zu verwalten (weitere MasterClients werden abgelehnt) und potentiell beliebig viele ListenerClients. Hier bestehen lediglich Beschränkungen hinsichtlich der Performance.

Für den MasterClient ist ein Authentifizierungsverfahren implementiert. Dieses stellt sicher, dass vom MasterClient empfangene Pakete auch tatsächlich vom MasterClient stammen. Dieses wird erreicht, indem Nachrichten, die der MasterClient an den Server übermittelt, ein MAC (Message Authentication Code) angehängt wird. Der Server kann dieses MAC verifizieren und damit sicherstellen, dass dieses Paket vom MasterClient stammt (siehe auch Abschnitt 4.2 „Login und Authentifizierung“).

Die ListenerClients müssen lediglich eine Netzwerkverbindung aufbauen und anschließend eine Login-Nachricht verschicken. Der Server wird diese immer akzeptieren. ListenerClients können sich jederzeit mit dem Server verbinden oder die Verbindung trennen. Sie können auch für mehrere Präsentationen am Server angemeldet bleiben.

---

<sup>6</sup> Autor: Sebastian Schwanewilms

### ***Präsentationsverwaltung***

Der Server verwaltet die vom MasterClient erhaltene Präsentation. Diese wird in Form von Bilddateien auf dem System abgelegt. Sobald der Server eine vollständige Präsentation vom aktuellen MasterClient zur Verfügung hat, wird er diese an alle ListenerClients weiter verteilen. Sollte sich zu einem späteren Zeitpunkt ein weiterer ListenerClient einloggen, wird der Server die Präsentation unmittelbar nach dem erfolgreichen Login an den ListenerClient übermitteln.

### ***Datenhaltung***

Der Server speichert alle erhaltenen Präsentationen im Dateisystem in Form von Bilddateien. In das gleiche Verzeichnis werden alle Audiodateien abgelegt, die von den ListenerClients (Mitschnitte von Redeanfragen) oder MasterClients (gesamter Mitschnitt der Präsentation) stammen.

### ***Kommandoweiterleitung***

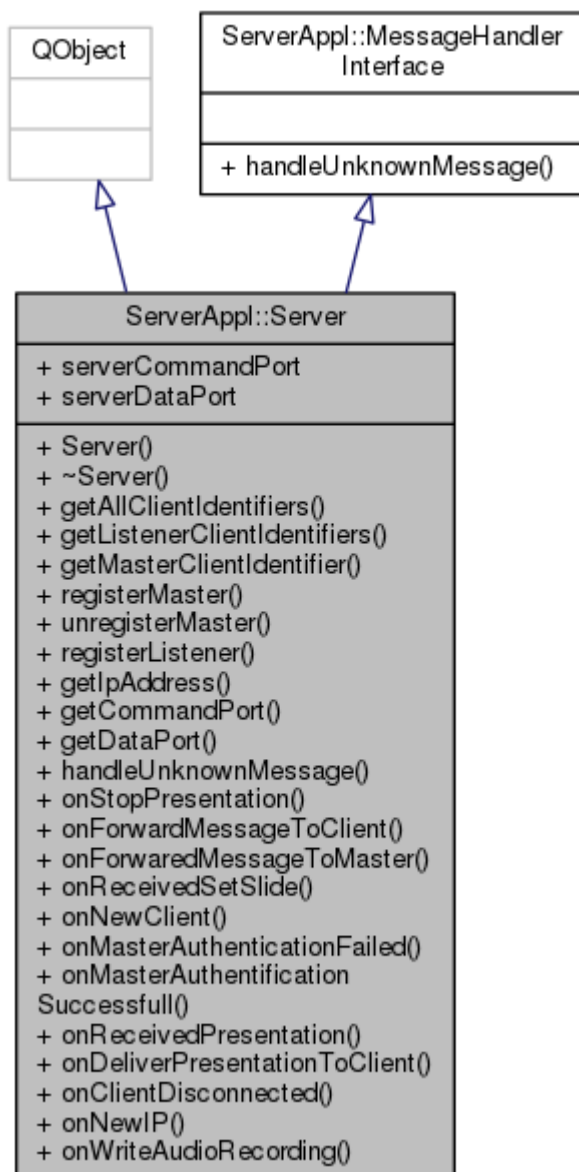
Der MasterClient übermittelt an den Server verschiedene Steuerkommandos bezüglich der Präsentation. Hierbei kann es sich um Kommandos zum Ändern der aktuell angezeigten Folie handeln, aber auch um die Annahme von Redeanfragen. Auch die ListenerClients können Kommandos an den Server senden. Abgesehen vom Loginprozess handelt es sich dabei jedoch lediglich um Redeanfragen.

Diese Kommandos werden ebenfalls in Form von XML-Nachrichten über das Netzwerk empfangen. Der Server untersucht diese empfangenen Kommandos und leitet sie entsprechend weiter. Gegebenenfalls verschickt er noch eine Empfangsbestätigung an den Absender.

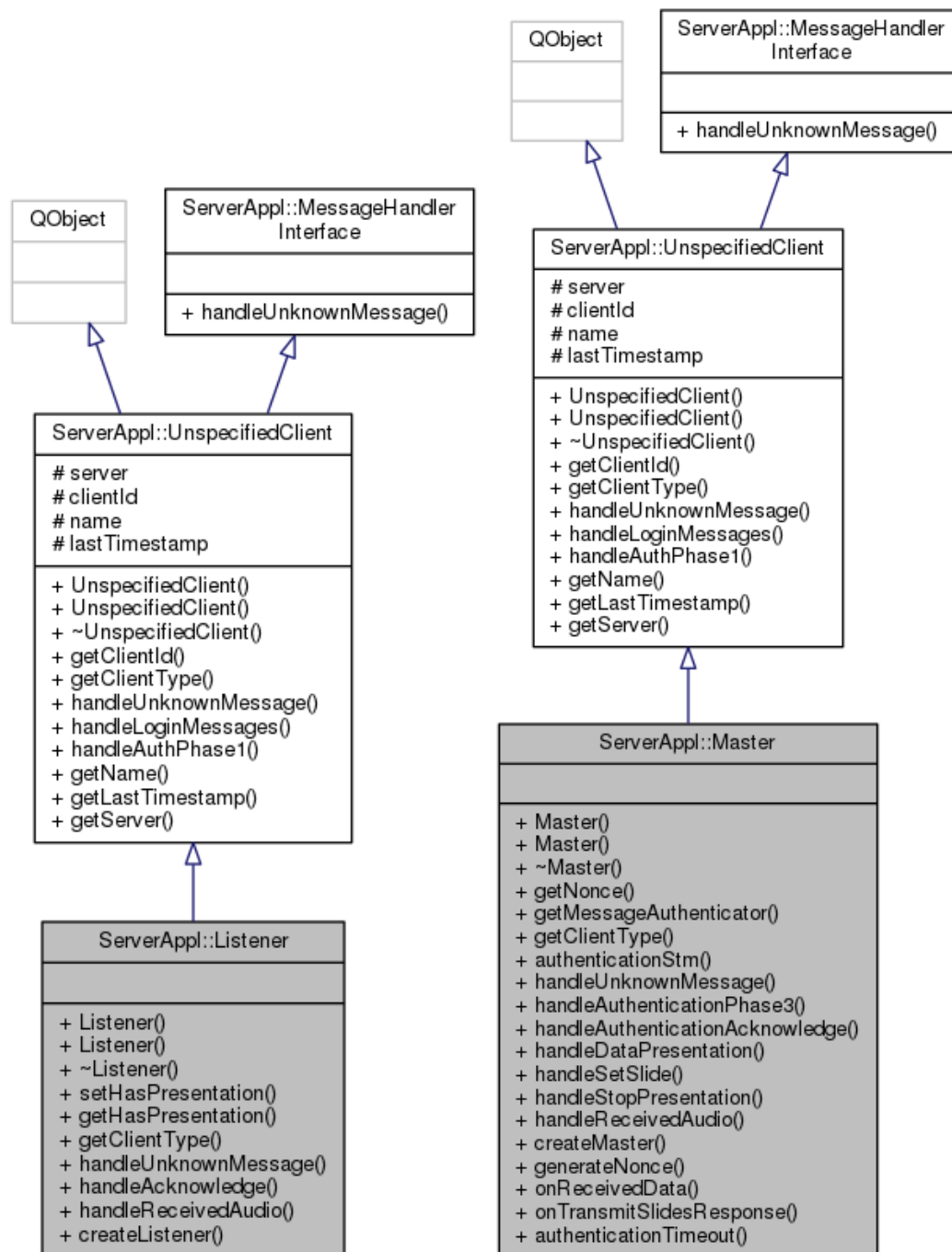
## Struktur

Die Server-Applikation besteht aus verschiedenen Klassen. Hier nun einige dieser Klassen als Abbildung in UML-Notation. Die Grafiken wurden mittels Doxygen generiert. Doxygen fügt in die Abbildungen aus Gründen der Übersichtlichkeit keine Parameter oder Rückgabetypen ein. Genauere Informationen hierzu können der Doxygen-Dokumentation entnommen werden.

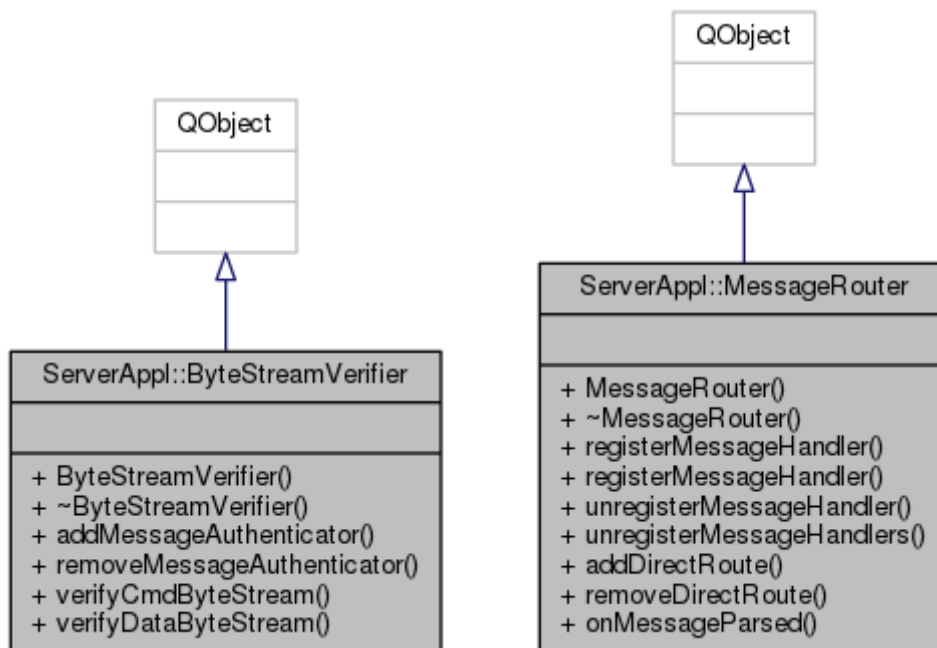
### Server



## Clients



## Messaging



## Beschreibung

Der Server stellt eine eigenständige Applikation dar. Die Konfiguration geschieht derzeit über den Quellcode. Es wäre jedoch unproblematisch, diese über eine Konfigurationsdatei bereitzustellen. Die Oberfläche zeigt lediglich die benötigten Daten (IP-Adresse und Port-Nummern für Kommando- und Datenpakete) für den Verbindungsaufbau an.

## Klasse Server

Intern stellt die Klasse `Server` die zentrale Komponente dar. Hier werden die verbundenen ListenerClients sowie der MasterClient verwaltet. Auch ist die Server-Klasse für das Abspeichern von Daten (Audiodateien, Präsentationen) verantwortlich. Von hier aus werden eingehende und verloren gegangene Verbindungen der Netzwerkschicht behandelt.

## Nachrichtenverarbeitung – Klassen `ByteStreamVerifier`, `XmlMessageParser`, `MessageRouter`

Eine von der Netzwerkschicht empfangene Nachricht durchläuft auf dem Server mehrere Klassen, bevor sie dem Empfänger zugestellt wird. Zunächst gibt die

Netzwerkschicht die Nachricht in Form eines Byte-Streams über ein Signal an den *ByteStreamVerifier* weiter.

Der *ByteStreamVerifier* prüft, ob die Nachricht verifiziert werden muss. Hierzu hält er eine Liste mit Clients vor, von denen empfangene Nachrichten überprüft werden müssen. Das Verfahren zur Verifizierung basiert darauf, dass dem Byte-Stream 28 Byte als MAC angehängen sind. Der *ByteStreamVerifier* kann anhand des eigentlichen Byte-Streams ebenfalls einen 28 Byte langen MAC bestimmen. Sollte der empfangene MAC mit dem selber berechneten MAC übereinstimmen, gilt der Byte-Stream als verifiziert. Bei der derzeitigen Implementierung wird dieses Verfahren lediglich für Nachrichten vom MasterClient angewendet. Der *ByteStreamVerifier* ist jedoch so implementiert, dass er auch die Nachrichten von anderen Clients überprüfen könnte.

Vom *ByteStreamVerifier* wird die Nachricht, nach wie vor in Form eines Byte-Streams, an den *XmlMessageParser* übergeben. Dieser generiert aus dem Byte-Stream ein Objekt der Klasse *Message*. Dieses wird mittels Signal an eine Instanz der Klasse *MessageRouter* weiter gegeben.

Die Klasse *MessageRouter* bietet zwei verschiedene Arten, um Nachrichten zu verteilen. Die erste Möglichkeit dient dazu Nachrichten weiterzuleiten, die auf dem Server zwischenverarbeitet werden müssen. Hierzu hält der *MessageRouter* eine Liste vor. Jedes Objekt (dessen Klasse von *MessageHandlerInterface* erbt) kann dort einen Befehlsnamen zusammen mit einer Client-Identifizierung (ClientId) und einem Pointer auf eine eigene Member-Funktion hinterlegen. Sollte ein Client mit der passenden ClientId das entsprechende Kommando an den Server übermitteln, wird der *MessageRouter* die entsprechende Member-Funktion des Objektes aufrufen. Diese Member-Funktion kann entweder ein Message-Objekt zurück liefern, welches an den Sender der ursprünglichen Nachricht zurück verschickt wird, oder es kann einen Null-Pointer zurück liefern. In diesem Fall wird keine Antwort-Nachricht an den ursprünglichen Sender geschickt.

Die zweite Möglichkeit zur Verteilung von Nachrichten wird nur angewendet, wenn der *MessageRouter* feststellt, dass in der Liste aus der zuvor beschriebenen Möglichkeit kein Eintrag für die empfangene Nachricht existiert. In diesem Fall wird das Message-Objekt untersucht. Es wird geprüft, ob an Hand der enthaltenen Empfängerangaben

eine direkte Weiterleitung der Botschaft an einen bestimmten Client möglich ist. Hierzu muss jedoch auch in einer anderen Liste ein Eintrag existieren. Diese zweite Liste enthält direkte Routen, das heißt, sie beschreibt mögliche Empfänger von Nachrichten.

### ***Nachrichtenversand***

Der Nachrichtenversand wird über die Klasse *XmlMessageWriter* durchgeführt. An beliebiger Stelle im System können Nachrichten mittels Signal an die Klasse *XmlMessageWriter* übergeben werden. Diese wird dann einen entsprechenden Byte-Stream erzeugen und ihn der Netzwerkschicht übergeben.

### ***Clientverwaltung – Klassen UnspecifiedClient, Master, Listener***

Zur Verwaltung der Clients (MasterClient und ListenerClients) existieren drei Klassen. Objekte der Klasse *UnspecifiedClient* werden verwendet, solange nach einer aufgebauten TCP/IP-Verbindung noch nicht klar ist, ob es sich um einen ListenerClient oder MasterClient handeln wird. Die Netzwerkschicht übergibt nach dem Aufbau eine eindeutige Identifizierung für den entsprechenden Client. Diese so genannte ClientId wird solange verwendet, bis die Verbindung auf Netzwerkebene getrennt wurde.

Der *MessageRouter* leitet alle Nachrichten mit Login-Befehlen an die entsprechenden Objekte der Klasse *UnspecifiedClient* weiter. An Hand des Login-Befehls lässt sich feststellen, ob es sich um einen Master- oder ListenerClient handelt. Es wird nun ein entsprechendes Objekt der dazu passenden Klasse (*Master* oder *Listener*) erzeugt. Das Objekt der Klasse *Server* wird über das neue Client-Objekt informiert. Es werden auch alle Einträge beim *MessageRouter* für das neue Client-Objekt eingefügt und die benötigten Signale und Slots werden verbunden.

Im Falle eines MasterClient übernimmt das neu erzeugte Objekt der Klasse *Master* die weiteren Schritte der Authentifizierung. Auch Steuerbefehle für die Präsentation werden von nun an über dieses Objekt ausgewertet.

Sollte der MasterClient eine Präsentation an den Server übermitteln, wird sie im *Master*-Objekt empfangen und an das *Server*-Objekt weiter gegeben. Dieses verteilt die Präsentation dann an alle ListenerClients.



### ***Klasse `Logger`***

Abschließend ist noch die Klasse *Logger* zu erwähnen. Hierüber ist es möglich, eine zentrale Log-Datei zu erstellen. Die Klasse *Logger* unterstützt Log-Nachrichten für das Debugging und für den normalen Betrieb. Um die entsprechenden Funktionen einfach zu verwenden, stehen entsprechende Makros bereit. Dies ermöglicht es auch, das Debug-Logging einfach zu deaktivieren.

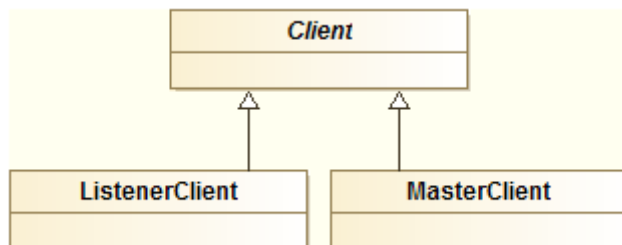
## 4.2 Client<sup>7</sup>

Die Applikation, die später dem Benutzer vorliegt wird im Allgemeinen als Client bezeichnet. In unserer Architektur haben wir den Client in zwei unterschiedliche Typen unterteilt:

*ListenerClient* - Client, der der Präsentation als Zuhörer beiwohnt.

*MasterClient* - Client, der die Präsentation leitet; der Vortragende.

Dabei ist zu beachten, dass beide Client Typen viele Gemeinsamkeiten haben. Ganz grob strukturiert kann man also den Client als Basisklasse mit den erbenden Klassen Master und Listener darstellen.



Da der Client ein komplexes Konstrukt mit einigen verwendeten Hilfsklassen darstellt wird im Folgenden auf die eigentlichen Funktionalitäten eingegangen. Dies erfolgt nach dem Standardablauf einer Präsentation. Anhand jeder Aktion wird Erläutert welche weiteren Konstrukte hinzukommen und wie diese im Client oder in einer seiner Ableitungen implementiert wurde, welche Entscheidungen gefällt wurden und was für Probleme auftraten. Weiterhin wird die entstandene Implementierung, falls nötig, kritisch kommentiert.

Vorbereitend sind die groben gemeinsamen Strukturen wie folgt aufgelistet:

*Netzwerkinterface* – das Netzwerkinterface ist grundsätzlich Identisch

*XML-Abstraktionsebene* – die XML-Ebene wird für beide Clienten gleich implementiert

*Präsentation* – der Status und die Datenhaltung der Präsentation wird auf beiden Seiten identisch implementiert

---

<sup>7</sup> Autor: Jan Zimmer

*Remote-Procedure-Call Framework* – die Remote-Procedure-Call Funktionalitäten werden von beiden Clients benötigt

*GUI Zugriffsmethoden* – viele Methoden zum GUI Zugriff werden von beiden Clients benötigt.

*HDMI Ausgabe* – die HDMI Ausgabe wird standardmäßig von allen Clients implementiert.

*Signal/Slots* – es gibt einige gemeinsame Signale und Slots; jeder Client ist gleichzeitig ein QObject.

*Hinweis:* Die die GUI betreffende Dokumentation und alle zugehörigen Felder sind gesondert in Abschnitt 4.5 enthalten und werden dort ausgearbeitet.

## Verarbeitung von Nachrichten

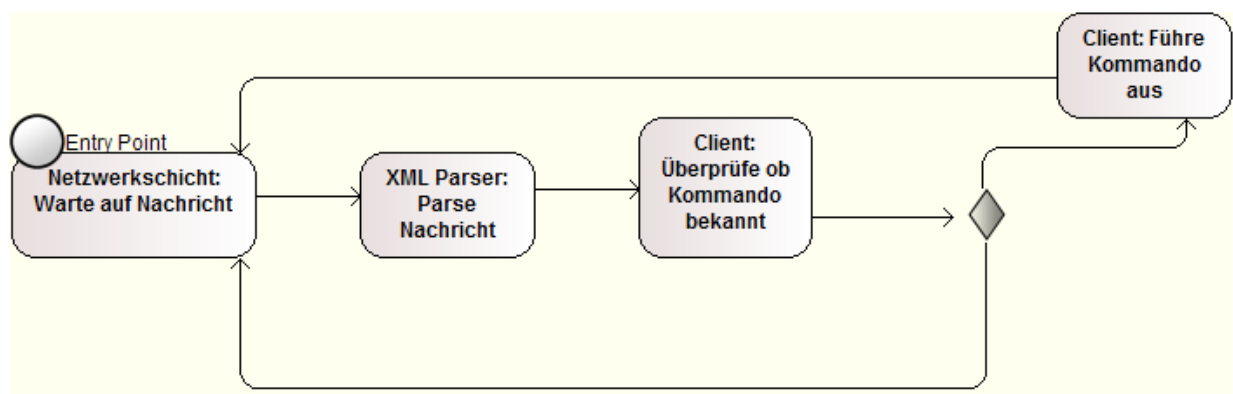
### Anforderung

Die Applikation muss möglichst einfach erweiterbar sein. Neue Funktionalitäten sollen schnell hinzugefügt werden können.

### Umsetzung

Alle Funktionen, die über das Netz auf anderen Instanzen ausgeführt werden sollen werden in einem einheitlichen Nachrichtenformat ausgetauscht (Details siehe Abschnitt 4.4)

Diese Nachrichten beinhalten das auszuführende Kommando sowie die dazu benötigten Parameter. Das Verarbeiten von Nachrichten ist von der Funktionalität in beiden Clients identisch. Es läuft wie folgt ab:



Zu Beginn wartet der Client auf eine Nachricht. Diese kommt im XML Format in der Netzwerkschicht an und wird anschließend vom XML Parser eingelesen und in ein Nachrichten Objekt überführt (siehe Abschnitt 4.4).

Das enthaltene Kommando wird überprüft und falls es bekannt ist ausgeführt. Ansonsten wird es stillschweigend ignoriert. Das so instanziierte Whitelisting von Kommandos trägt hier zur Stabilität der Applikation bei (eine vollständige Auflistung aller Kommandos, siehe Anhang)

Um zu überprüfen, ob ein Kommando unterstützt wird und dieses gegebenenfalls auszuführen enthält jeder Client folgendes Feld:

**Felder (Client)**

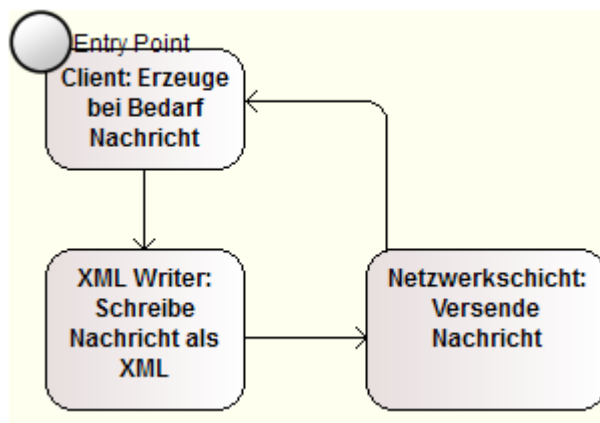
*Abbildung von Unterstützten Kommandos zu Funktionen* – enthält alle Kommandos mit Namen und einen Funktionszeiger auf die Entsprechende auszuführende Funktion. Wird als QMap implementiert, die String (Name des Kommandos) auf Funktionszeiger abbildet.

*XMLParser/Writer* – wandelt Nachrichten ins/vom Message-Objekt um. Muss sowohl für Daten als auch für Kommandos vorhanden sein.

Weiterhin gibt es einen Slot im Client, der Nachrichten empfängt und die angeforderte Funktionalität gegebenenfalls ausführt.

Um eine strikte Entkopplung zu gewährleisten, werden die einzelnen Schritte per Signal/Slot Mechanismus verbunden. Man stelle sich das so vor, dass jeder Pfeil in der obigen Statemachine einem Signal an die nächste Instanz gleichzusetzen ist, die ihrerseits einen passenden Slot zur Verfügung stellt. So werden die einzelnen Komponenten entkoppelt und müssen sich nicht gegenseitig „kennen“.

Das Versenden von Nachrichten erfolgt quasi analog:



Auch hier werden Daten jeweils über Signal/Slot-Mechanismen zwischen den Einzelnen Instanzen ausgetauscht.

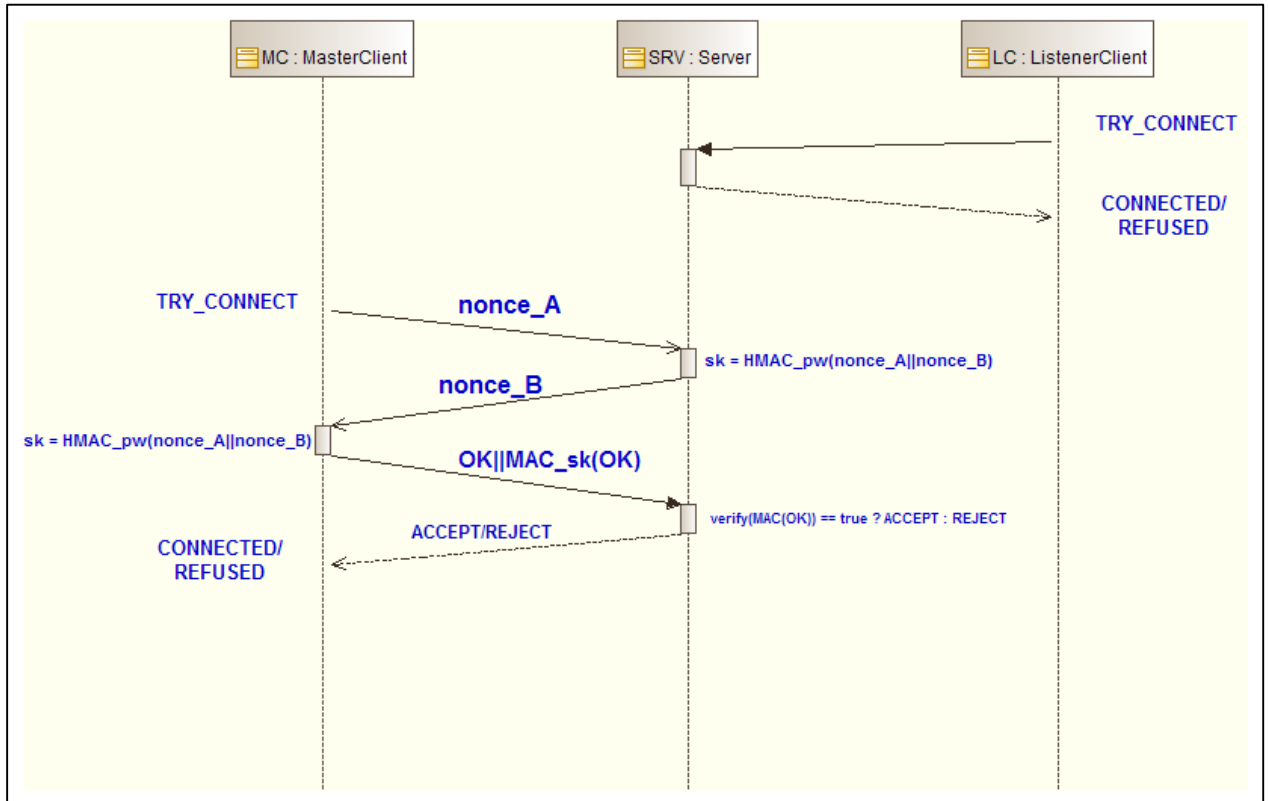
#### Kritischer Kommentar

Der Typ von Funktionen, der als Funktionszeiger in der Liste der ausführbaren Kommandos abgespeichert wurde, wurde am Anfang des Projektes im Client explizit als eine Memberfunktion eben jenes definiert. Im Laufe des Projektes wurde festgestellt, dass dies eine falsche Entscheidung war, da so alle Funktionen explizit im Client oder einer seiner Kindklassen implementiert werden mussten. Dadurch wurde Funktionen als Wrapper doppelt geschrieben, die auch direkt in den verwendeten Hilfsobjekten implementiert sind. Eine Lösung für dieses Problem wäre ein Interface-ähnliches Konstrukt zu implementieren um auch Funktionen anderer Klassen registrierbar zu machen.

Dieses Vorgehen wurde im Server auch verfolgt. Im Client hätte dieser Schritt zu viel Zeit in Anspruch genommen.

## Login und Authentifizierung

Um mit dem Server kommunizieren zu können, muss sich jeder Client mit dem Server verbinden. Dies geschieht getrennt für Master und Listener unterschiedlich.



### Anforderung

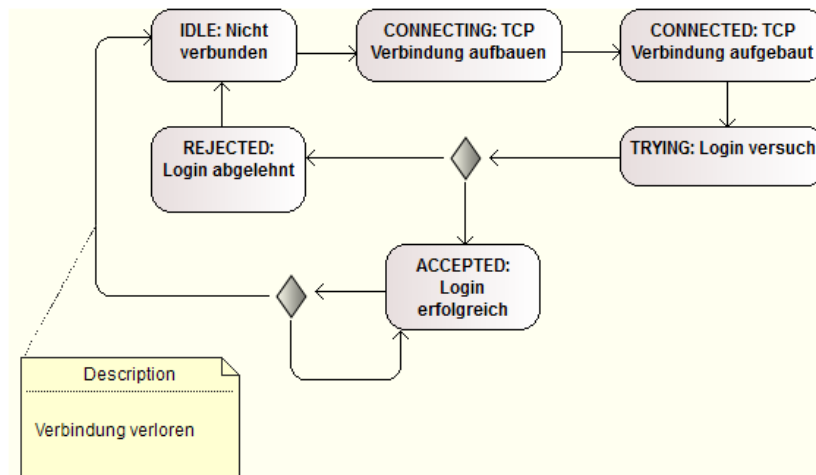
Der Anwender wird für den Login Prozess in Kenntnis über die Server IP sowie die beiden Ports zur Daten- und Kommandoübertragung gesetzt. Anschließend soll es ihm möglichst einfach möglich sein, sich zum Server zu verbinden.

### Umsetzung

Ein Listener Client logged sich in den Server ein, indem es eine Nachricht mit dem Login-Wunsch an den Server gesendet wird. Dieser antwortet mit einem Status OK falls er den Client akzeptiert und ansonsten mit einem Status NOT OK. Der Client verhält sich entsprechend der Antwort (siehe obige Abbildung).

Zu beachten ist hierbei, dass zu unterscheiden ist zwischen der TCP-Verbindung die vom Server grundsätzlich akzeptiert wird und dem logischen Login, bei dem der Client auf dem Server registriert wird.

Der Client enthält ein Feld, das seinen momentanen Status beschreibt. Der detaillierte Ablauf eines Logins ist in folgendem Diagramm dargestellt:



Weiterhin wird ein Feld implementiert, das den eindeutigen Identifier für den jeweiligen Client enthält.

#### Felder (Client)

*Login-Status* – beinhaltet Login-Status

*Identifier* – beinhaltet eindeutigen Identifier. Wird von Server vergeben.

#### Methoden (Client)

*Login* – initiiere den Login

#### Master

##### Anforderung

Da der Master zur Steuerung der Präsentation eingesetzt wird, ist eine Anforderung, dass der Server Master nur unter der Eingabe eines Passwortes akzeptiert.

Weiterhin sollte die Kommunikation zwischen Master und Server authentisch stattfinden, um Manipulation der Nachrichten zu verhindern. Dabei müssen die Nachrichten explizit NICHT vertraulich sein.

## Umsetzung

Der grundsätzliche Login Ablauf wird beibehalten. Zusätzlich wird als einfache Maßnahme der Authentifizierung ein Einfaches Schlüssel-Austausch-Protokoll implementiert, das wie folgt definiert ist:

Phase 0:	Der Veranstalter setzt ein Passwort für den Server fest und verteilt dieses an die Vortragenden.
Phase 1:	Master sendet eine ausreichend lange Zufallszahl ( <i>nonce_A</i> ) an den Server.
Phase 2:	(a) Server sendet eine ausreichend lange Zufallszahl ( <i>nonce_B</i> ) an den Master. (b) Server berechnet $sk = MAC_{PASSWORT}(nonce\_A    nonce\_B)$ einen Sitzungsschlüssel.
Phase 3:	(a) Master berechnet $sk = MAC_{PASSWORT}(nonce\_A    nonce\_B)$ einen Sitzungsschlüssel. (b) Master sendet $OK    MAC_{sk}(„OK“)$ an Server zurück.
Phase 4:	Master verifiziert den MAC der Nachricht und akzeptiert den Login bei erfolgreicher Verifizierung.
(Siehe auch obige Abbildung).	

**Definition:** Wir verwenden das etablierte Verfahren HMAC als MAC. Als MAC wird eine symmetrische Signatur bezeichnet. Nur Inhaber des verwendeten Schlüssels können diese Erzeugen und verifizieren. Der MAC wird dabei über eine Nachricht berechnet. Wird der MAC einer Nachricht erfolgreich verifiziert, so bedeutet das, dass die Nachricht nicht manipuliert wurde.

Im Folgenden wird kurz beschrieben welchen Vorteil das Protokoll hat und wie es grob funktioniert.

Das Protokoll verhindert, dass das gewählte Passwort im Klartext über das Netzwerk übertragen wird, da sich ein Man-in-the-Middle Angreifer (also ein Angreifer, der sich



zwischen den Master und den Server auf die Netzwerkverbindung setzt und diese belauschen und manipulieren kann) dieses so einfach per Mitlesen verschaffen könnte. Anstelle dessen *beweist* der Master, dass er im Besitz des korrekten Passwortes ist, indem er aus diesem einen Wert ableitet/mit diesem Berechnung durchführt.

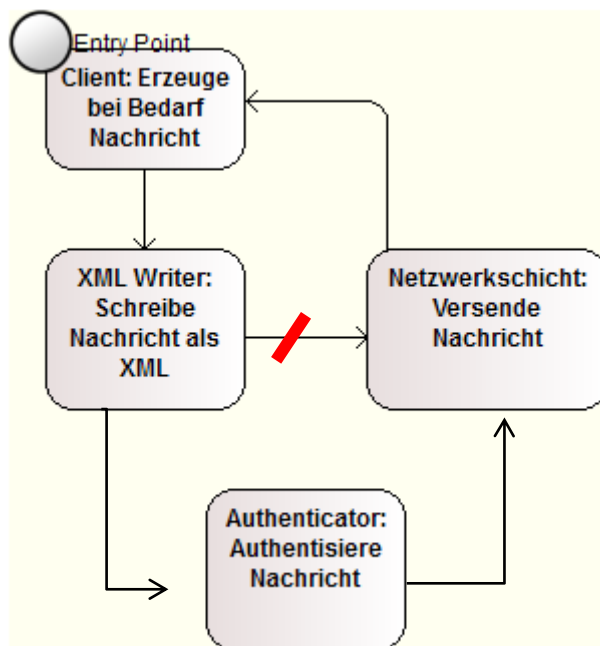
Genauer leiten beide Parteien, da sie im Besitz des Passwortes sind, einen Schlüssel für eine Sitzung ab (Phase 1, 2, 3(a)). In Phase 3(b) berechnet der Master einen MAC über die Nachricht „OK“ und verwendet dazu den berechneten Schlüssel. Wenn der Server diesen MAC in Phase 4 mit dem berechneten Schlüssel erfolgreich verifizieren kann, so bedeutet das für ihn, dass der Master im besitzt des Korrekten Passwortes ist – dem Server wurde dies soeben *bewiesen*.

Als weiteres Detail ist anzumerken, dass dadurch, dass beide Parteien jeweils eine Zufallszahl erzeugen, ein Angreifer nicht einfach alte Zufallszahlen verwenden kann. Auch wird in jeder Sitzung ein neuer Schlüssel verwendet.

Im weiteren Verlauf der Kommunikation gilt immer:

Sendet der Server eine Nachricht  $M$ , so hängt er dieser, sobald er erfolgreich authentifiziert ist, IMMER ein MAC berechnet mit dem Sitzungsschlüssel an (also  $M || \text{MAC}_{sk}(M)$ ). Der Server akzeptiert nur Nachrichten deren MAC er erfolgreich verifizieren kann.

Das Senden der Nachricht ändert sich also wie folgt:



Es wird einfach durch Umkonfigurieren der Signale erreicht.

Zum Halten des Authentifizierungsstatus wird ein Feld im MasterClient implementiert. Weiterhin werden eine Methode zum Setzen des verwendeten Passwortes und ein Feld für das Passwort und den Sitzungsschlüssel implementiert.

Außerdem wird eine Klasse MessageAuthenticator implementiert, die bei eingabe von Binärdaten und einem Schlüssel ein MAC berechnet und anhängt. Dabei erfolgt der Datenaustausch mit dieser mit Hilfe von Signal/Slot.

#### **Felder (MasterClient)**

*Auth-Status*

*Passwort*

*Sitzungsschlüssel*

*Nachrichten Authentisierer*

#### **Methoden (MasterClient)**

*SetzePasswort*

### Kritischer Kommentar

Dieses Protokoll ist bewusst simple Definiert. Es bietet besseren Schutz als ein einfaches Austauschen des Passwortes. Weiterhin sind Nachrichten vom Master an den Server authentisiert. Folgendes wäre für eine produktive Implementierung zu beachten und wurde nicht Implementiert:

- Der Server muss seine Nachrichten ebenfalls authentisieren, damit Nachrichten vom Server an den Master nicht manipuliert werden können. (Anmerkung: Dies ließe sich leicht in den Master integrieren)
- Der Server und der Master müssten Vorrichtungen treffen um die Gleiche Nachricht nicht erneut zu erhalten. Dies könnte Anhand des bereits vorhandenen Zeitstempels der Nachricht gelöst werden. Neue Nachrichten müssten also immer einen späteren Zeitstempel als die letzte gültige Nachricht haben.
- Die Kommunikation zu den Listnern ist bis jetzt nicht abgesichert. Manipulierte Nachrichten werden von diesen akzeptiert. Hier sollte überlegt werden ein standardisiertes Protokoll wie TLS zu verwenden um die Kommunikation zwischen Server und Client abzusichern.

### **Vorbereiten der Präsentation**

Bevor überhaupt eine Präsentation gehalten werden kann muss diese vorbereitet und verteilt werden. Dazu bedarf es einer Klasse, die alle Informationen über die Aktuelle Präsentation enthält und die auf entsprechende Änderungswünsche reagieren kann. Die Klasse *Praesentation* erfüllt diese Aufgabe.

### ***Präsentation***

#### Anforderung

Die Präsentation enthält konsekutive Folien fixer Anzahl. Durch die Präsentation kann navigiert werden. Dabei ist zu jeden Zeitpunkt definiert auf welcher Folie man sich befindet. Eine Präsentation hat einen eindeutigen Identifier.

## Umsetzung

Bevor die Klasse *Praesentation* spezifiziert und umgesetzt werden kann, muss zuerst festgelegt werden wie eine Präsentation genau auszusehen hat.

### *Spezifikation des Präsentationsformates*

Da das Projekt in sehr straffem Zeitrahmen vollendet werden muss, fällt die Entscheidung hier nach dem Prinzip „Keep it simple and stupid“. Eine Präsentation ist von Natur aus eine Folge von Abbildungen. Dieses Schema wird beibehalten und auf eine konsekutive Folge von Bilddateien abgebildet. Diese müssen einen fest kodierten Dateinamen haben. Eine Zahl von maximal 1000 Folien erscheint hier als ausreichend. Der Dateiname wird also spezifiziert zu:

*3-stellige Foliennummer (0-Präfix), Start bei Folie 0. (Beispiel: 000.jpg, 011.jpg)*

Es sind vorerst nur JPEG Bilder zugelassen.

Zusammengehörige Folien bilden eine Präsentation. Sie werden dazu in einem gemeinsamen Ordner platziert. Weiterhin ist zu bemerken, dass eine Lücke in der Nummerierung NICHT erlaubt ist. Tritt eine solche auf gilt grundsätzlich: die letzte Folie vor der Lücke schließt die Präsentation ab.

### *Klasse Praesentation*

Die Klasse *Praesentation* muss folgende Felder beinhalten:

#### **Felder (*Praesentation*)**

*Aktuelle Folie* – Nummer der Aktuellen Folie (definierter Startzustand: -1)

*Anzahl der Folien* – Gesamtzahl der Folien (definierter Startzustand: 0)

*Präsentations-ID* – Identifier der Präsentation, muss eindeutig sein. Wird zufällig nach folgendem Muster erstellt: aktuelles Datum (DD\_MM\_YYYY) + „\_“ + 8-stellige, Hexadezimale Zufallszahl.

*Liste von Folien* – beinhaltet die Folien. Wird implementiert durch eine Liste von URLs die auf Folien im Basisverzeichnis der Präsentation verweisen.

*Status Indikator* – zeigt an, ob die Präsentation bereits läuft.

Sie muss folgende Methoden implementieren:

#### **Methoden (Praesentation)**

*Setze Folie* – setzt die Folie auf eine bestimmte Nummer. Das erste setzen einer Folie wird als Start der Präsentation gewertet.

*Hänge Folie an* – hängt eine Folie an die Präsentation an.

*Ausgabe von aktueller Folie und Anzahl der Folien* – gibt geforderte Information zurück.

*Stop* – Präsentation stoppen.

*Reset* – Präsentation zurücksetzen. So wird ein Neuanlegen der Präsentation überflüssig.

Weiterhin soll die Klasse folgende Funktionalitäten bereitstellen um die Struktur nicht unnötig kompliziert zu machen. Die Klasse soll sich selber in eine Nachricht serialisieren können und sich aus einer Nachricht initialisieren. Diese zwei Datenhaltungsmethoden werden ebenfalls in der Klasse implementiert.

Anzumerken ist, dass die Klasse Präsentation als QObject implementiert wird und so das Signal/Slot Prinzip von QT verwenden kann.

Die Klasse stellt folgende Signale bereit um mit der Außenwelt automatisiert zu kommunizieren:

#### **Signale (Praesentation):**

*Folie geändert* – enthält die Folie (oder eine Referenz)

*Läuft* – enthält eine Angabe ob die Präsentation läuft oder gestoppt wurde.

*Am parsen* – enthält Informationen ob die Präsentation noch geparsed wird oder ob sie bereits vollständig eingelesen wurde.

#### *Verwendung von Praesentation im Client*

Jeder Client beinhaltet genau eine Instanz einer Präsentation. Dabei füllt der Master seine Präsentation über die GUI mit Folien, die im Anschluss an die Zuhörer verteilt und dort eingelesen wird.

## **Felder (Client)**

*Präsentation* – enthält das Präsentation-Objekt

### Kritischer Kommentar

In der Klasse *Praesentation* wurde zum signalisieren eines Folienwechsels zwei Signale implementiert. Das eine enthielt dabei das bereits geladene und konvertierte Bild. Das zweite Signal enthält nur noch die URL auf die Folie. Bedingt wurde dies durch die verschiedenen Ansprüche einzelner Komponenten und insbesondere das Zeitversetzte auftreten dieser. Dabei wurde zuerst die Ausgabe in der GUI implementiert, die das Bild als Binärdaten akzeptiert und erst im Nachhinein die Ausgabe auf externen Displays, die wiederum die URL benötigt und daraufhin das Bild selber lädt.

Als Verbesserung müsste hier frühzeitig ermittelt werden, was für technische Anforderungen die überlappenden Komponenten haben. Dies ließ sich allerdings durch den knappen Zeitplan nicht synchron durchführen.

### **Verteilung der Präsentation**

Die Verteilung der Präsentation erfolgt einmalig am Anfang und in vollem Umfang, um die Netzlast (und die damit verbundene Latenz) während der Präsentation zu minimieren. Sie wird dazu vom Master an den Server und von dort an alle Zuhörer verteilt.

### **Ändern der Folie**

Das Ändern der Folie stellt eine Funktionalität dar, die in unserer Struktur wie folgt Platz findet. Um möglichst synchrones und einheitliches Arbeiten zu ermöglichen werden immer alle Kommandos über den Server verteilt und von dort aus weitergeleitet. Dies betrifft auch das setzen einer neuen Folie.

Die Aktion lässt sich also in zwei Teile aufteilen:

1. Erzeugen und Stellen der Anfrage an den Server: Diese Aktion wird nur im Master implementiert, da ausschließlich dieser dazu befugt ist.
2. Empfangen und auswerten eines Folienwechsels: Dies wird gleichermaßen auf Client wie auf Server implementiert.

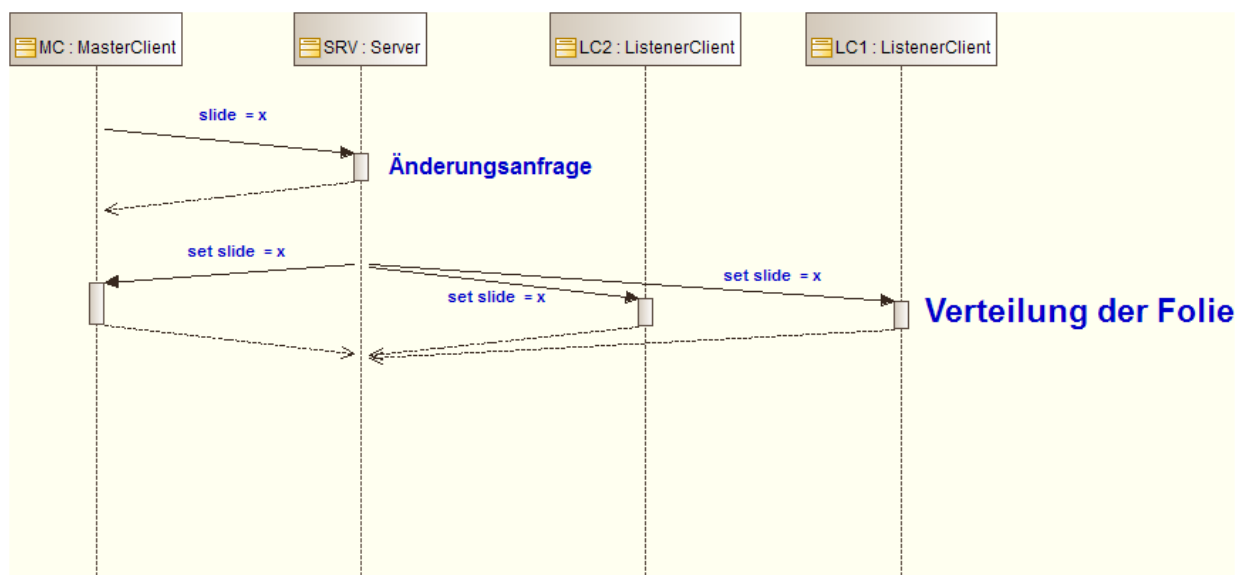
Der Folienwechsel wird nur anhand einer Foliennummer signalisiert. Die Folie selber wird dann jeweils lokal geladen. So erübrigt sich das redundante Versenden großer Datenmengen, insbesondere beim Zurückgehen auf vorherige (bereits einmalig ausgelieferte Folien).

Um den Folienwechsel möglichst flexibel zu gestalten, werden zwei Methoden im MasterClient implementiert:

#### Methoden (MasterClient)

*Wechsel Folie mit Offset* – Die Folie wird mit einem Offset zur aktuellen Folie geändert. Es wird so das Vor- und Zurückschalten der Folien vereinheitlicht. Eine „Fast-Skip“ Funktionalität ließe sich schnell nachrüsten.

*Wechsel Folie absolut* – Die Folie wird auf eine absolute Foliennummer gesetzt. So lässt sich die Präsentation auf Anfrage schnell und Präzise steuern.



Der interne Ablauf ist anhand des Sequenzdiagramms gegeben. Der Master fragt einen Folienwechsel am Server an. Dieser verteilt diese Anfrage an ALLE Clients. Diese wiederum setzen die Folie.

#### Kritischer Kommentar

Aufgrund mangelnder Kenntnisse in C++ und QT wurden viele Funktionalitäten sowohl im Client als auch in der Hilfsklasse Praesentation implementiert. Dabei wurden gerade Signale einfach 1:1 weitergeleitet, da in QML später nur der Client eingebunden

wurde und so nur dessen Signale abgefangen werden konnten. Auch Funktionen, die direkt aus der GUI aufgerufen werden (Setzen der Folie), wurden jeweils in beiden Klassen implementiert, wobei der Client größtenteils als Wrapper fungiert.

Dies widerspricht dem Grundmodell der funktionalen Trennung und der Redundanz Vermeidung. In einer überarbeiteten Version kann dieser Umstand umgangen werden, indem man die Präsentation als QTMemeber implementieren würde und den Typ explizit in QML bekannt macht. So könnte man direkt Funktionen der Präsentation von QT aus ansprechen.

Diese Wrapper Eigenschaft des Clients zieht sich durch das gesamte Projekt und ist dem oben genannten Umstand geschuldet.

### **Stoppen der Präsentation**

Die Präsentation kann vom Master explizit gestoppt werden. Dabei ist anzumerken, dass auch diese Anfrage über den Server an alle Clients verteilt wird.

### Anforderung

Der Master soll die Präsentation stoppen können. Dies wird bei allen Teilnehmern signalisiert.

### Umsetzung

Die Aktion lässt sich in zwei Teile aufteilen:

1. Erzeugen und Stellen der Anfrage an den Server: Diese Aktion wird nur im Master implementiert, da ausschließlich dieser dazu befugt ist.
2. Empfangen und auswerten des Stopp Kommandos: Dies wird gleichermaßen auf Client wie auf Server implementiert. Dabei reagiert der jeweilig Client, indem er eine vordefinierte Folie anstelle der vorherigen Präsentationsfolie einblendet. Der Einfachheit halber wird die selber Folie eingeblendet, wie vor Beginn der Präsentation.



Es kommen zusätzlich folgende Eigenschaften hinzu:

#### **Methoden (MasterClient)**

*Stoppe Präsentation anfragen* – der Server wird angefragt die Präsentation zu stoppen.

#### **Methoden (Client)**

*Stoppe Präsentation* – setze Folie und stoppe Präsentation.

### **Redeanfragen**

Die Redeanfrage stellt ein zentrales Feature der Anwendung dar. Sie soll einen geregelten Ablauf der Kommunikation zwischen dem Vortragenden und den Zuhörern ermöglichen.

#### Anforderung

Während laufender Präsentation, soll es dem Zuhörer möglich sein, den Vortragenden auf eine Frage hinzuweisen. Dem Vortragenden soll dies explizit signalisiert werden. Der Vortragende soll einfach auf die Anfragen reagieren können.

#### Umsetzung

##### *Redeanfrage*

Zunächst muss eine Klasse *Redeanfrage* implementiert werden (Hinweis: da sich der Begriff „Redeanfrage“ im Sprachgebrauch des Teams schnell als prägnanter Begriff etabliert hat, wurde explizit darauf verzichtet den Klassennamen ins Englische zu übersetzen), die die Redeanfrage als solche repräsentiert. Dazu müssen folgende Felder enthalten sein:

#### **Felder (Redeanfrage)**

*Status* – es muss zu jeder Zeit klar sein in welchem Status sich die Präsentation befindet.

*Identität des Anfragenden* – die Anfrage muss eindeutig einem Anfragenden zugeordnet werden.

Weiterhin müssen noch Methoden implementiert werden, die den Status der Anfrage ändern. Der Einfachheit halber wurden auch Funktionen implementiert, die das Überführen der Anfrage in das jeweilige Nachrichtenformat erlauben.

Als nächstes muss ein geeigneter Ablauf der Redeanfrage spezifiziert werden. Es wurden folgende Annahmen gemacht:

1. Jeder Zuhörer kann nur eine Anfrage gleichzeitig stellen.
2. Um das Beantworten der Fragen so einfach wie möglich zu gestalten, sollen Fragen immer in der Reihenfolge beantwortet werden, in der sie versendet wurden.
3. Eine einmalig gestellte Frage soll nicht einfach zurückgenommen werden können. Als Ersatz wird bei Annahme der Anfrage nachgefragt, ob diese noch aktuell ist, und gegebenenfalls abgebrochen.
4. Wird die Anfrage von beiden Seiten bestätigt, so wird auch beim Zuhörer Audiomaterial mittgeschnitten, um später eine gute Qualität der Frage für den Zusammenschnitt bereitzustellen.
5. Eine einmalig von beiden Seiten angenommene Anfrage kann nur vom Master beendet werden. (Hintergrund: Da die Anfrage auf beiden Seiten aufgezeichnet wird, soll der Vortragende die Kontrolle über die Aufnahme behalten, um so höchst mögliche Qualität zu gewährleisten).

#### **Hintergrund:**

Im Folgenden wird kurz das angenommene Szenario beschrieben, wie eine Frage abläuft:

Der Zuschauer möchte eine Frage stellen – er signalisiert dies dem Vortragenden. Dieser ist momentan noch im Redefluss – die Frage wird gespeichert (es kann explizit eine (längere) Zeitspanne vergehen). Während dessen können andere Zuhörer weitere Anfragen stellen. Der Vortragende kommt an einen Punkt, an dem er bereit ist Fragen zu beantworten. Da die Clients vom Vortragenden nicht zuordenbar sind, beantwortet er die Fragen in der Reihenfolge des Eintreffens. Dies ist auch die faireste Behandlungsweise. Er bestätigt die Anfrage. Entweder, die Frage hat sich seit der Anfrage bereits erledigt (der Zuhörer lehnt dann ab), oder die Frage wird diskutiert und anschließend beendet.

Es werden weiterhin noch zwei Szenarien betrachtet:

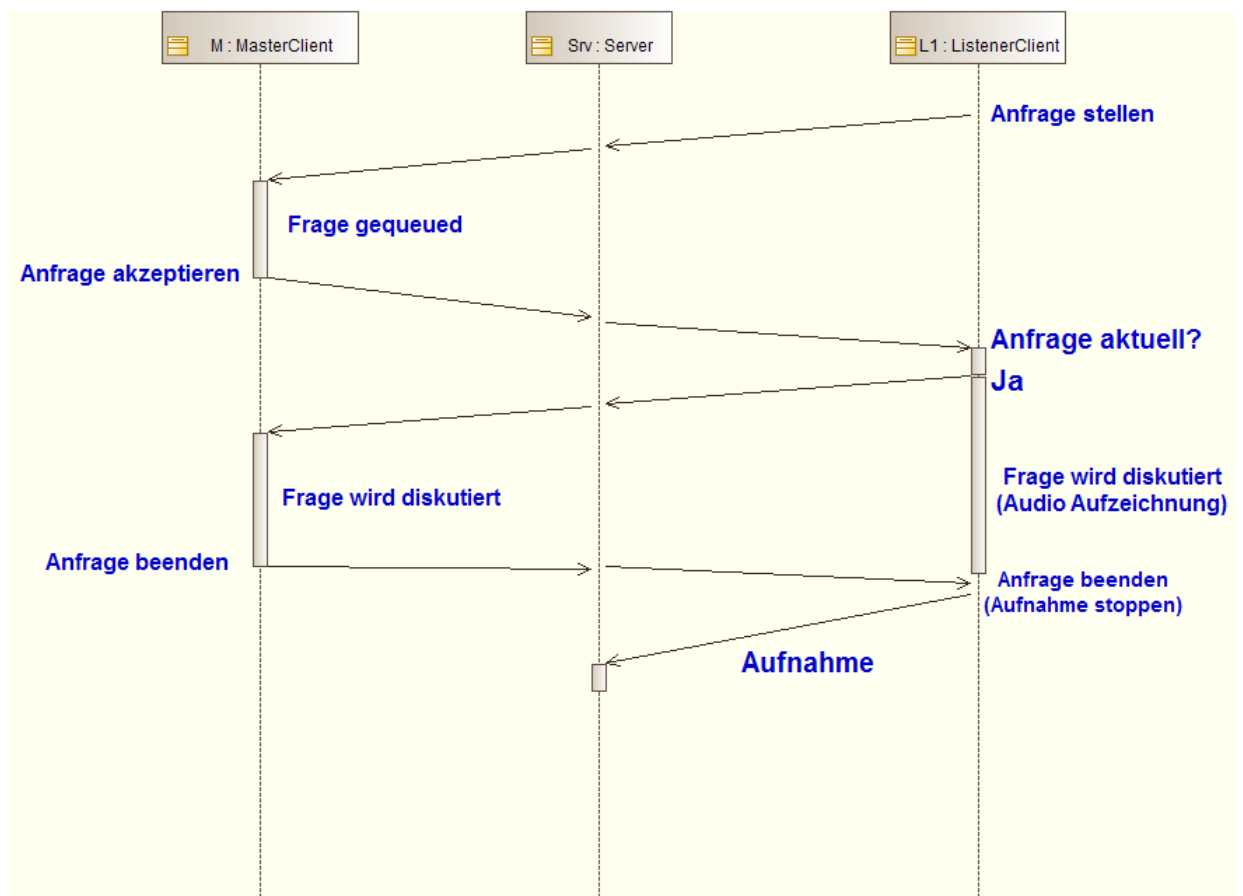
1. Der Vortragende fragt, ob überhaupt noch offene Fragen sind und löscht gegebenenfalls ALLE Anfragen.
2. Der Vortragende möchte nicht gestört werden – ihm muss eine Funktionalität bereitgestellt werden, automatisch alle Anfragen abzulehnen.

Es kommen also folgende Annahmen hinzu:

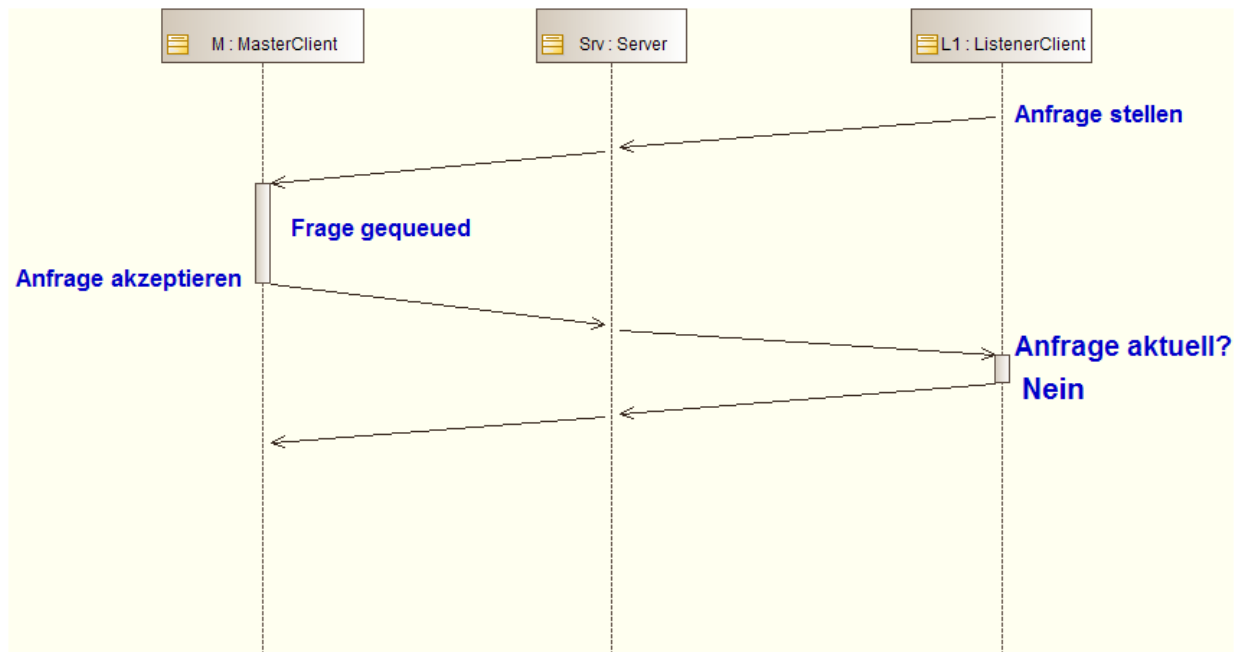
6. Anfragen werden nur einzeln angenommen oder kollektiv abgelehnt.
7. Anfragen können unterdrückt werden.

Modelliert man alle möglichen Abläufe als Sequenzdiagramm wo ergibt sich folgende:

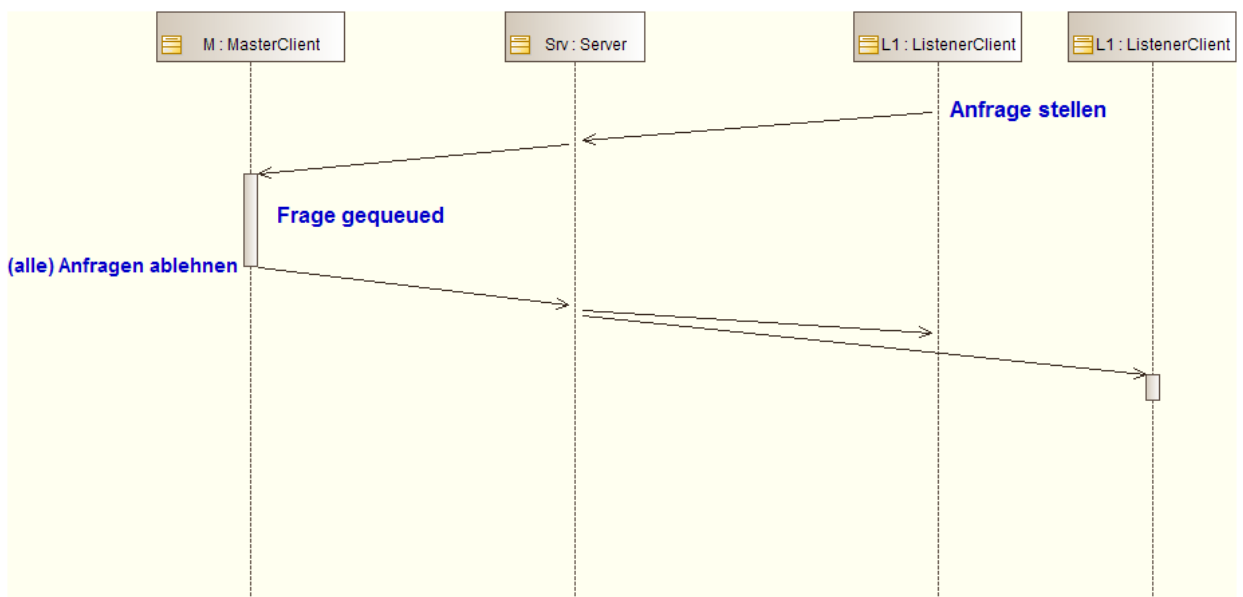
Beide Stimmen zu:



Frage ist nicht mehr aktuell:



Alle Fragen werden abgelehnt:



### *RedeanfrageQueue*

Der Ablauf erfordert beim Master ein Feld, das wie eine Queue funktioniert und die Redeanfragen aufnimmt. Da auf diese Queue gegebenenfalls threaded zugegriffen wird (Netzwerk und GUI gleichzeitig), muss die Klasse threadsafe implementiert werden. Dies erfolgt über einen Mutex.

Es müssen folgende Methoden bereitgestellt werden:

#### Methoden (RedeanfrageQueue)

*Hinzufügen* – wenn ein Zuhörer noch keine Anfrage gestellt hat, wird diese hinten angehängt.

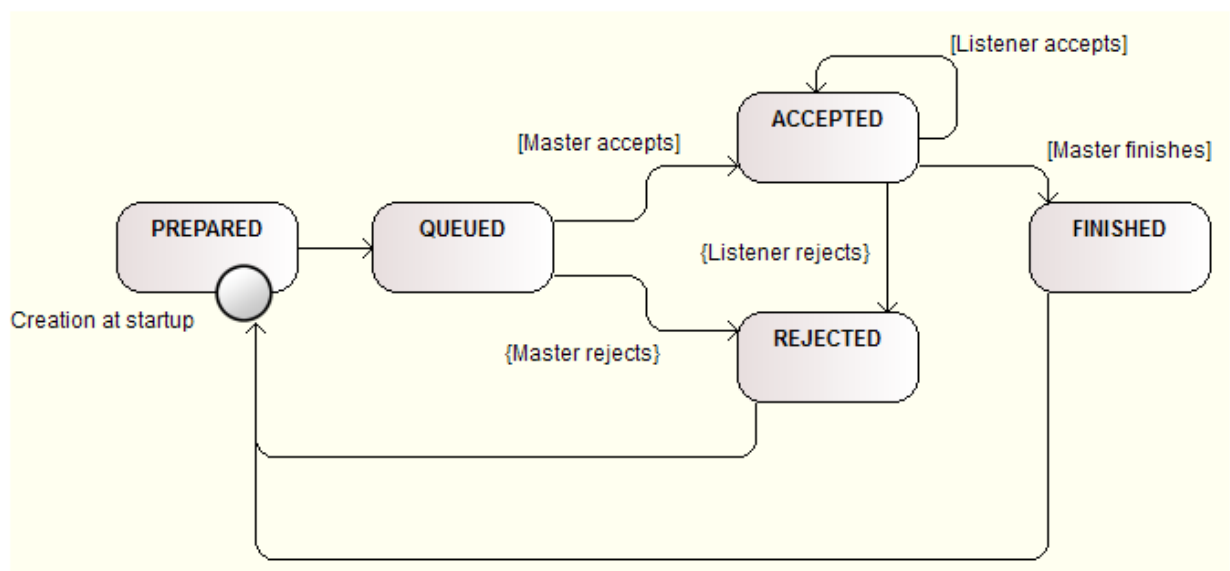
*Entnehmen* – das älteste Element wird der Liste entnommen.

*Leeren* – die Warteschlange wird geleert.

Um zu signalisieren, ob ein Element überhaupt hinzugekommen ist, wird ein Signal implementiert, das anzeigt, dass sich die Größe geändert hat und wie die neue Größe ist.

#### Umsetzung im ListenerClient

Der ListenerClient enthält ein Feld, das seine Redeanfrage enthält. Diese wird während der Laufzeit nicht gelöscht, sondern der Status der Redeanfrage wird geändert. Dabei ist der Statusablauf wie folgt definiert:



Jedes Mal, wenn der Zuhörer eine Frage stellt, wird diese in den Status Queued gesetzt. Von da an ist der Ablauf bereits oben beschrieben. Weiterhin wird eine Methode implementiert, die die Anfrage auslöst und von der GUI aus erreichbar sein muss.

### Umsetzung im MasterClient

Der MasterClient implementiert 2 Felder:

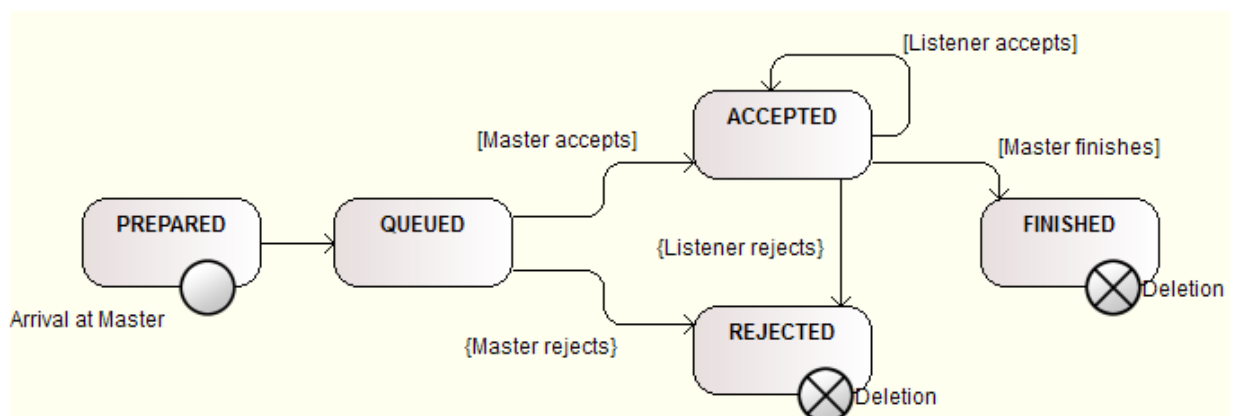
#### Felder (MasterClient)

*Aktuelle Anfrage* – verweist auf die aktuell akzeptierte Anfrage

*Anfragenliste* – eine Liste, die alle Anfragen beinhaltet (RedeanfragenQueue)

Weiterhin werden Methoden zum Annehmen, alle Ablehnen und zum Umschalten zwischen „stumm“ und „akzeptieren“ implementiert.

Eine Abfrage enthält der Einfachheit halber die gleichen Zustände. Diese werden allerdings nur genau einmal nach folgendem Modell durchlaufen und die Anfrage wird dann wieder gelöscht:



### Kritischer Kommentar

Um die Stabilität des Ablaufs zu gewährleisten, müssten Timeouts definiert werden, da sonst Verklemmungen auftreten können, wenn ein Zuhörer nicht auf eine akzeptierte Anfrage antwortet.

### Audioaufzeichnung

Ein weiteres zentrales Feature ist das Aufzeichnen von Audiomaterial während der Präsentation. Dabei wird beim Vortragenden durchgängig aufgezeichnet während Zuhörer nur während einer Frage aufgezeichnet werden.

### Anforderung

Der Vortrag soll aufgezeichnet werden. Dazu werden sowohl der Vortragende als auch die Zuhörer bei Fragen aufgezeichnet.

### Umsetzung

Die genaue Implementierung wird in Abschnitt 4.6 beschrieben.

In den Clients wird dazu jeweils ein Feld für den Audiorecorder implementiert. Zusätzlich wird eine Methode implementiert, die das Audiomaterial nach Aufzeichnung an den Server sendet.

### Kritischer Kommentar

Da das Cascades Framework bereits eine Audiorecorderklasse anbietet, wurde trotz eigener Implementierung am Ende der angebotene Audiorecorder verwendet und in QML eingebunden. An dieser Stelle gab es kurze Kommunikationsschwierigkeiten, die zu doppelter Arbeit geführt haben.

### **Externe Ausgabe**

Die Applikation soll Folien auf einem per HDMI angeschlossenen Display ausgeben.

### Anforderung

An mehreren Clients soll die Ausgabe der Folien auf externen Displays möglich sein, um so eine verteilte Darstellung ohne Kabelverlegen zu ermöglichen. Die Ausgabe soll konfigurierbar sein.

### Umsetzung

Details zur Ansteuerung des Displays, siehe Abschnitt 4.7.

Im Client wird das Objekt zur Ansteuerung als Feld eingebunden und im Konstruktor initialisiert. Bei Start der Präsentation wird die erste Folie ausgegeben. Da das Objekt selber keine Slots anbietet, wird im Clients ein Wrapperslot implementiert, der bei Wechsel der Folie entsprechend die Folie auf dem Display ändert.

Die Anforderung der Konfigurierbarkeit wurde explizit nicht umgesetzt. Diese wird bereits dadurch abgebildet, dass eine Ausgabe nur stattfindet, wenn auch ein Kabel angeschlossen ist.

## **Gestensteuerung**

Der Vortragende soll anhand von Gesten durch die Präsentation navigieren können.

Dazu wurde eine Instanz der Gestensteuerung (siehe Abschnitt 4.8) als Feld im MasterClient implementiert. Weiterhin wurden Wrappermethoden zum ein- und ausschalten der Gestensteuerung für die GUI implementiert.

## **Kritischer Kommentar (Client)**

Wie oben beschrieben, ist der Client ein komplexes Konstrukt. In der Entwicklungsphase waren begründet durch Zeit- und Kenntnismangel in C++/QT teilweise Funktionalitäten nur grob spezifiziert und mussten schnell umgesetzt werden, um Tests mit anderen Komponenten durchzuführen. Auf Grund dessen lässt sich in der Abschließenden Implementierung feststellen, dass der Client zu Teilen einer „gewachsenen“ Struktur und nicht einer Implementierung nach Spezifikation entspricht.

Diese Problematik hätte sich nur vermeiden lassen, wenn entweder deutlich mehr Zeit zur Verfügung gestanden hätte, oder auf viele der zusätzlichen Features bewusst verzichtet worden wäre und nur das einfache Grundkonstrukt implementiert worden wäre. Da die Applikation aber eher den Charakter eines „Proof-of-Concept“ für uns hatte, war die Implementierung der Features für uns von ausschlaggebender Wichtigkeit.

Im Umkehrschluss ließ sich anhand dieser Problematiken ein besseres Lernergebnis erzielen, da man sich mit komplexeren Strukturen auseinandersetzen musste.



## 4.3 Netzwerkschicht<sup>8</sup>

### Anforderungen

An die Netzwerkschicht der Applikation wurden verschiedene Anforderungen gestellt. Als Transferprotokoll sollte TCP genutzt werden. Die Netzwerkschicht selber, sollte keine von ihr übertragenen Daten auswerten oder analysieren. Vielmehr sollte sie in der Lage sein, Daten zuverlässig zwischen Client und Server auszutauschen und nur Signale auslösen, sobald Daten komplett übertragen wurden und zur Verfügung stehen. Zusätzlich sollten alle mit dem Server verbundenen Clients in einer Liste o.ä. gespeichert und verfügbar gemacht werden.

### Funktionalität

Die Netzwerkschicht der PräsiBERT-Applikation besteht grundsätzlich aus zwei Typen von Socket-Klassen. Einer *ClientSocket* Klasse für die Nutzerseite und einer *ServerSocket* Klasse für die Server-Applikation. Zusätzlich benutzt die *ServerSocket* Klasse eine weitere Klasse *ConnectedClient*, von der für jeden neu verbundenen Client ein neues Objekt erzeugt und in einen eigenen Thread verschoben wird.

Beide Sockets sind asynchron implementiert und können mittels Signals und Slots Daten versenden und ausgeben. Als Kommunikationsprotokoll wird TCP verwendet.

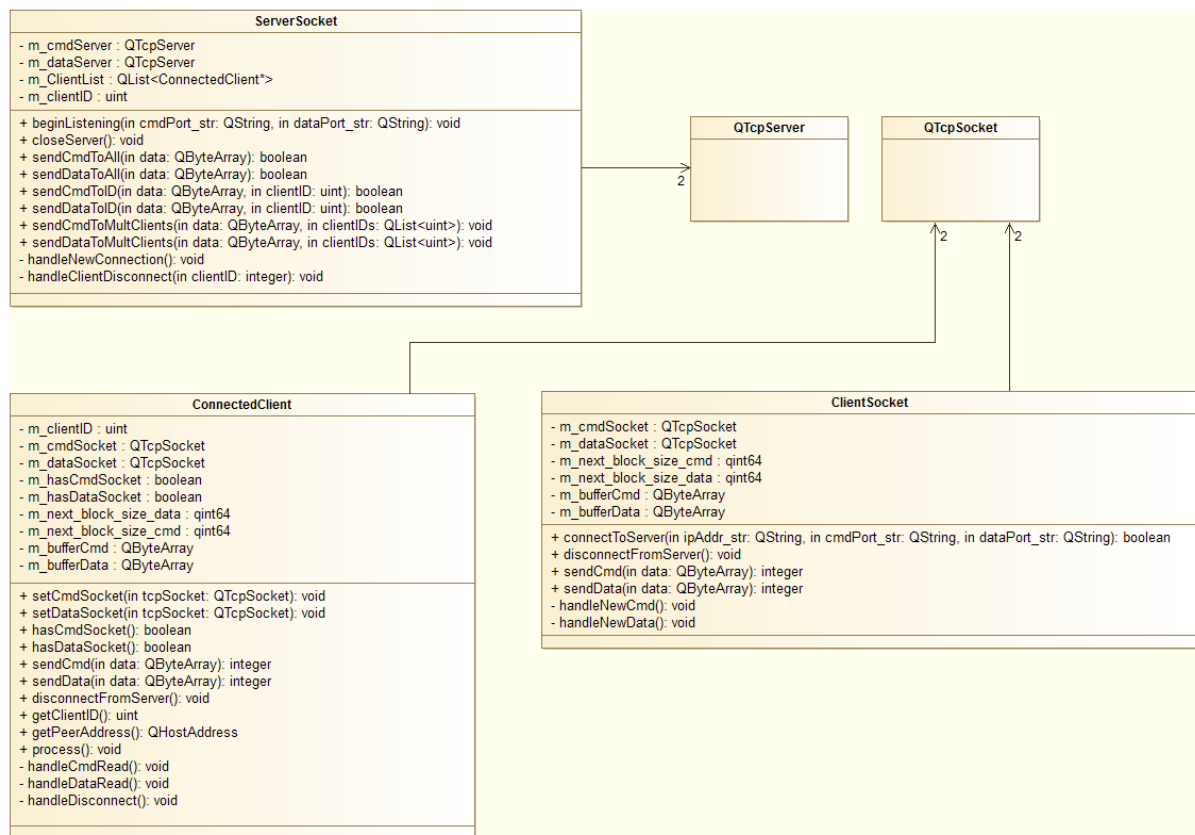
Sowohl *ServerSocket* als auch *ClientSocket* besitzen jeweils zwei Arten von Sockets für die Datenübertragung. Einen Socket zum Senden und Empfangen von Kommandos und einen Socket zum Übermitteln von Datenpaketen. So soll verhindert werden, dass Kommandos durch zu lange Datenpakete zu spät oder gar nicht empfangen werden. Für jede der Übertragungsarten wird ein eigener Port verwendet.

---

<sup>8</sup> Autor: Niklas Kröger

## Struktur

Das folgende UML-Diagramm zeigt die grundlegende Struktur der Netzwerkschicht.



Neben den im UML-Diagramm dargestellten Methoden und Variablen, stellen die Klassen noch verschiedene Signale zur Verfügung:

ClientSocket	ServerSocket	ConnectedClient
connectedToCmdServer()	newIP()	newCmd()
connectedToDataServer()	clientDisconnected()	newData()
receivedCmd()	stoppedServer()	disconnected()
receivedData()	receivedCmdFromClient()	finished()
lostConnection()	ReceivedDataFromClient()	

Weitere Informationen zu den Funktionsweisen der einzelnen Methoden sind der Doxygen-Dokumentation im Anhang zu entnehmen.

## Beschreibung

### ***Die ClientSocket Klasse***

Mithilfe der *ClientSocket* Klasse lassen sich Kommando- und Daten-Socket für einen Client erstellen. Sie initialisiert zwei Arten von Sockets (Kommandos und Daten) vom Typ *QTcpSocket*.

#### Herstellen einer Verbindung

Um sich mit einem Server zu verbinden wird die Funktion *connectToServer()* verwendet. Als Parameter werden sowohl die IP-Adresse als auch die beiden Ports für Daten und Kommandos im *QString* Format übergeben. Die Methode stellt dann eine Verbindung zu dem Server an der gewünschten Adresse her. Gewöhnlich verbindet sich der Kommando-Socket zuerst, allerdings ist der Server so konstruiert, dass es unabhängig ist, welcher Socket sich zuerst verbindet. Bei einem Fehler liefert die Methode den Wert **false** zurück. Wird die Verbindung erfolgreich hergestellt, so liefert die Methode den Wert **true** zurück.

#### Senden von Daten:

Über die Funktionen *sendCmd()* und *sendData()* lassen sich Kommandos oder Daten im *QByteArray* Format an den Server schicken, mit dem der Client verbunden ist. Ein 32 Bit langer Integer mit der Datenwortlänge in Byte wird an den Anfang der Daten gesetzt. Die Funktion liefert als Rückgabewert die Anzahl der versendeten Daten in Byte.

#### Empfangen von Daten:

Die Klasse stellt die Signale *receivedCmd()* und *receivedData()* zur Verfügung. Je nachdem welche Art von Daten empfangen wird, wird eines der Signale ausgelöst. Als Übergabeparameter werden die empfangenen Daten in *QByteArray* Format übergeben. Die *ClientSocket* Klasse achtet darauf, dass diese Signale nur ausgelöst werden, wenn die Daten komplett übertragen wurden und verfügbar sind.

#### Verbindung zum Server trennen:

Über den Slot *disconnectFromServer()* kann die aktuelle Verbindung zum Server geschlossen werden.

### Verbindung zum Server verlieren:

Die Klasse stellt das Signal *lostConnection()* zur Verfügung. Sollte die Verbindung eines Sockets zum Server abbrechen oder beendet werden, werden beide Verbindungen geschlossen und dieses Signal ausgelöst.

### **Die ServerSocket Klasse**

Mithilfe der *ServerSocket* Klasse lässt sich ein Socket für die Server Applikation erstellen. Sie initialisiert zwei Arten von Servern (Kommandos und Daten) vom Typ *QTcpServer*. Für jeden Client, welcher sich mit dem Server verbindet, wird ein Objekt der Klasse *ConnectedClient* erstellt. Diese Klasse enthält wiederum zwei Sockets über welchen der Server mit dem jeweiligen Client kommunizieren kann.

### Auf Verbindungen von Clients warten

Über den Slot *beginListening()* wird der Server initialisiert und für eingehende Verbindungen vorbereitet. Als Parameter müssen die beiden Ports für Kommandos und Daten im *QString* Format übergeben werden. Nachdem die Routine abgearbeitet ist, wird von der Funktion das Signal *newIP()* ausgelöst, welches die aktuelle IP des Servers im *QString* Format zur Verfügung stellt. Konnte keine korrekte IP ermittelt werden, so wird *Localhost* als neue IP ausgegeben.

### Neue Clients verbinden sich mit dem Server

Die Klasse stellt den Slot *handleNewConnection()* zur Verfügung, welcher mit den Signalen *newConnection()* der beiden Server-Sockets verbunden ist. Verbindet sich ein neuer Client mit dem Server, überprüft die Routine welcher der beiden Server-Sockets das Signal ausgelöst hat. Dementsprechend wird ein neues *QTcpSocket*-Objekt, welches die jeweilige Verbindung zum Client darstellt, erzeugt.

Ist unter der IP-Adresse der neuen Verbindung schon ein *ConnectedClient* mit einem der Sockets verfügbar, wird das *QTcpSocket* Objekt an das *ConnectedClient* Objekt mit dieser IP-Adresse übergeben. Dabei wird darauf geachtet, dass der vorhandene Socket nicht ersetzt wird. Ist unter der IP-Adresse der neuen Verbindung noch kein *ConnectedClient* verfügbar, wird ein neues Objekt der Klasse *ConnectedClient* erstellt und der Socket übergeben. Für jedes der erstellten *ConnectedClient* Objekte wird ein eigener Thread angelegt. Wird der Thread gestoppt oder der Server geschlossen, wird der Thread über verschiedene Verbindungen von Signals und Slots terminiert.

Alle Clients werden innerhalb der *ServerSocket* Klasse in einer Liste (*QList*) gespeichert. Jeder Thread bzw. Socket erhält eine individuelle ID, über die jeder Client eindeutig identifizierbar ist.

#### Nachrichten von Clients erhalten

Jeder verbundene Client kann das Signal *newCmd()* oder *newData()* mit den Daten im *QByteArray* Format und der ID als Parameter aussenden. Diese werden in der Routine *handleNewConnection()* mit dem Signal *receivedCmdFromClient()* bzw. *receivedDataFromClient()* der *ServerSocket* Klasse verbunden, sodass diese bei neuen Nachrichten automatisch ausgesendet werden.

#### Nachrichten an Clients senden

Über die Funktion *sendCmdToAll()* oder *sendDataToAll()* mit den Daten im *QByteArray* Format als Parameter lassen sich Kommandos oder Daten zu allen, mit dem Server verbundenen Clients, senden. Über die Funktion *sendCmdToID()* oder *sendDataToID()* mit den Daten im *QByteArray* Format und der *clientID* als Parameter lassen sich Kommandos oder Daten zu einem bestimmten, mit dem Server verbundenen, Client senden. Über die Funktion *sendCmdToMultClients()* oder *sendDataToMultClients()* mit den Daten im *QByteArray* Format und den IDs der Clients in einer *QList* als Parameter lassen sich Kommandos oder Daten zu mehreren, mit dem Server verbundenen, Clients senden.

#### Die Verbindung zu einem Client verlieren

Jeder verbundene Client sendet das Signal *disconnected()* mit der ID als Parameter aus, wenn die Verbindung einer der Sockets unterbrochen worden sein sollte. Über die Routine des Slots *handleClientDisconnect()* entfernt der Server den Client aus seiner Liste und löst das Signal *clientDisconnect()* mit der Client ID als Parameter aus.

#### Den Server beenden

Über die Funktion *closeServer()* lässt sich der Server stoppen. Mit dem Aufruf der Funktion wird jede Verbindung zu einem Client geschlossen. Anschließend wird der Server gestoppt und das Signal *stoppedServer()* ausgelöst.

### ***Die ConnectedClient Klasse***

Die Klasse *ConnectedClient* wird genutzt, um für jeden Verbundenen Client ein eigenes Objekt mit einem *QTcpSocket* zu erzeugen. Ein Objekt dieser Klasse erhält kein *QObject* als *parent*, damit sich das Objekt in einen eigenen Thread schieben und ausführen lässt. Jedes Objekt der Klasse erhält über seinen Konstruktor eine *Client ID*, über die der Client eindeutig zugeordnet werden kann.

#### Neuer Client verbindet sich mit dem Server:

Nach dem Erzeugen des Objekts und Verschieben in einen eigenen Thread, wird durch den Aufruf von *start()* im Thread der Slot *process()* aufgerufen. Anschließend werden über die Methoden *setCmdSocket()* und *setDataSocket()* die Sockets des Verbunden Clients entsprechend gesetzt.

#### Neue Daten von Client sind verfügbar:

Wenn neue Kommandos oder Daten von einem Socket zur Verfügung stehen, wird von diesem Socket das Signal *readyRead()* ausgelöst. Über die Slots *handleCmdRead()* und *handleDataRead()* werden die Daten gelesen und mit einem 32 Bit Integer überprüft, ob diese Vollständig vorhanden sind. Anschließend wird das Signal *newCmd()* bzw. *newData()* ausgelöst. Dieses hat die gelesenen Daten im *QByteArray* Format und die Client ID als Parameter.

#### Daten zum Client senden:

Über die Funktion *sendCmd()* bzw. *sendData()* mit den zu sendenden Daten im *QByteArray* als Parameter lassen sich Kommandos oder Daten zu dem Client senden. Vor die eigentlichen Daten wird ein 32 Bit Integer mit der gesamten Datenwortlänge gesetzt. Die Funktion übergibt die Anzahl der gesendeten Daten in Byte als Rückgabeparameter.

#### Die Verbindung zum Client beenden:

Die Funktion *disconnectFromServer()* beendet die Verbindung zwischen Client und Server. Es wird anschließend das Signal *finished()* ausgelöst, um den Thread zu beenden in dem der Client läuft.

#### Die Verbindung zum Client verlieren:

Sollte die Verbindung zu einem der Sockets des Client unterbrochen werden, wird von dem jeweiligen Socket das Signal *disconnected()* ausgelöst. Über den Slot *handleDisconnect()* werden anschließend beide Sockets geschlossen, das Signal *disconnected()* mit der Client ID als Parameter ausgelöst und das Signal *finished()* ausgelöst.

#### **4.4 Kommunikation/Nachrichtenformat<sup>9</sup>**

Die Anforderung an das Netzwerkinterface von Seiten der Entwickler war es, eine voll transparenten Zugriff auf das Netzwerk im Sinne von „Sende Nachricht X an Partei Y“ ohne weitere Detailkenntnisse über das Netzwerk zu ermöglichen.

Hierzu gehört auch die Spezifikation eines Einheitlichen Nachrichtenformats. Zu unterscheiden ist hierbei ein Nachrichtenformat während der Entwicklung (also eine Nachrichten Klasse) und das Format, mit dem eine Nachricht ins Netzwerk versendet wird.

##### **Nachrichtenformat auf Netzwerkebene**

Zur Diskussion des Nachrichtenformats auf Netzwerkebene standen folgende Formate:

- XML
- JSON
- Proprietäres Binärformat

---

<sup>9</sup> Autor: Jan Zimmer

Für die diskutierten Formate ergaben sich folgenden Vor- und Nachteile:

### ***XML***

(+)

- bei überschaubarer Nachrichtenstruktur gut (Menschen-) lesbar
- QtCore enthält XMLParser bereits (keine eigene Implementierung notwendig)
- weit verbreiteter Standard (Verwendung sinnvoll, da Aufwand zur Entwicklung/Einarbeitung später „wiederverwendet“ werden kann)

(-)

- Relativ hoher Overhead bedingt durch Darstellung als Text
  - o ABER: Hauptlast liegt bei Bild- und Audioübertragung, deshalb fällt Overhead nicht so ins Gewicht.

### ***JSON***

(+)

- Geringerer Overhead als XML
- Gut lesbar (JavaScript Object Notation)
- QtCore enthält JSON Parser (QT5)

(-)

- JSON standardisiert kein Datumsformat → muss man selber spezifizieren und auch parsen (Probleme bei Zeitstempeln zu erwarten)

### ***Proprietäres Binärformat***

(+)

- Geringster Overhead bei guter Strukturierung

(-)

- Eigene Spezifikation, Parser nötig (viel initialer Entwicklungsaufwand)



## Umsetzung

Nach diesen Vor- und Nachteilen fiel die Wahl auf das XML Format, da der Entwicklungsaufwand überschaubar bleibt und während des Debugging einfach lesbare Nachrichten erzeugt werden. Weiterhin ist das XML-Format dem beauftragten Entwickler gut bekannt und es wird so Einarbeitungszeit gespart.

Weitergehend muss ein allgemeines Nachrichtenformat in XML-Form spezifiziert werden. Dies kann grundsätzlich als XML-Schema getan werden. Da dies allerdings aufwändig ist und für die Anwendung nur wenige Felder benötigt werden wird hiervon abgesehen und stattdessen ein schematisches Nachrichtenformat skizziert und die zugehörigen Felder beschrieben:

```
<message>
  <header>
    <sender>
      A
    </sender>
    <receiver>
      B
    </receiver>
    <timestamp>
      01.01.2000 13:37:01.337
    </timestamp>
  </header>
  <command>
    do_something
  </command>
  <parameters>
    <parameter name=parameter_1 type=string>
      <data>Dies ist ein String!</data>
    </parameter>
  </parameters>
</message>
```

Eine Nachricht (*message*) besteht aus dem Header (*header*), einem Kommando (*command*) und den Parametern (*parameters*).

**Header:**

Der Header beinhaltet den Absender (*sender*) und der Empfänger (*receiver*) der Nachricht sowie einen Zeitstempel (*timestamp*) im Format: dd.MM.yyyy hh:mm:ss.zzz

**Kommando:**

Enthält das auszuführende Kommando als String.

**Parameterliste:**

Enthält eine ungeordnete Liste von Parametern (*parameter*). Dabei ist die Anzahl der Parameter gegeben von 0 bis unendlich.

**Parameter:**

Ein Parameter enthält die Attribute Name (*name*) und Typ (*type*).

*Name:*

Der Name des angegebenen Parameters als String.

*Typ:*

Der Typ eines Parameters darf folgende Werte annehmen:

*integer* – Das enthaltene Datum ist eine natürliche (ganze) Zahl.

*decimal* – Das enthaltene Datum ist eine Zahl mit Nachkommastellen.

*string* – Das enthaltene Datum ist eine Zeichenkette.

*date* – Das enthaltene Datum ist ein Datum in folgendem Format:  
dd.MM.yyyy hh:mm:ss.zzz

*b64* – Das enthaltene Datum ist ein Base64-Encodierter Binärstrom.

Ein Parameter enthält sein Payload (*data*), der dem Typ entsprechend eingefügt wird.

Zur Umsetzung dieses Nachrichtenformates wurden zwei Klassen *XMLMessageParser* und *XMLMessageWriter* implementiert die intern QT-eigene XML Parser und Writer verwenden. Diese Klassen sorgen dafür, dass das skizzierte Schema eingehalten wird.

Das spezifizierte Format wird folgend auch als Klasse auf Entwicklerseite modelliert und bildet das Nachrichtenformat ab.

### **Nachrichtenformat auf Entwicklerebene**

#### ***Umsetzung***

Eine Nachricht wird als Klasse *Message* modelliert. Dabei enthält die Klasse folgende Attribute:

#### **Felder (Message)**

*Parameterliste* - Liste aller enthaltenen Parameter

*Typliste* - Liste des Typs eines Parameters korrespondierend zu Parametern

*Absender/Empfänger*

*Timestamp*

Weiterhin werden folgende Methoden angeboten:

#### **Methoden (Message)**

*addParameter(String name, T value)* - Fügt einen Parameter anhand seines Typs T automatisch in beide Listen ein. Binärdaten werden dabei automatisch Base64 kodiert abgelegt.

*timestamp* - Setzt den Zeitstempel der Nachricht.

Implementiert wurden die beiden Parameterlisten als QMap, die jeweils einen String, den Namen des Parameters, auf einen QVariant (ermöglicht abspeichern eines beliebigen Datentyps inklusive Stringkonvertierung) im Falle des Parameterinhalts oder auf einen weiteren String (den Typ) im Falle der Typliste enthält. Weiterhin wurde darauf geachtet, dass Parameter beim Hinzufügen nur einmal in der Liste stehen. Die Typisierung der Parameter ist im obigen Abschnitt der XML spezifikation nachzulesen

und wurde jeweils innerhalb der QVariant als QT-Pendant umgesetzt (int, double, QString, QDateTime, QByteArray).

Zusätzlich wurde eine automatische Überprüfung auf Base64 eingebaut, sodass sowohl Base64 kodierte als auch rohe Binärdaten mit einer Methode eingefügt werden können.

Außerdem ist zu beachten, dass Message als QObject implementiert wurde um das Signal/Slot Modell von QT nutzen zu können. Ein weiterer Vorteil ist, dass enthaltene Daten nicht kopiert werden, sondern dass nur mit Referenzen auf das Objekt gearbeitet wird, was insbesondere bei großen Nachrichten Performancevorteile bietet.

Nachteil wiederum ist, dass man sich an entsprechender Stelle darum kümmert, erzeugte Objekte auch wieder zu löschen. Hierauf wurde an den jeweiligen Stellen hingewiesen und die Entwickler wurden unterrichtet.

## 4.5 GUI<sup>10</sup>

Die GUI auf Client Seite soll möglichst einfach und intuitiv sein. Sie wurde Vollständig in QML umgesetzt. Um ein leichtes integrieren des Fachkonzepts in die GUI zu ermöglichen, wurde der jeweilige Client als Objekt direkt in QML instanziiert. So ließ sich einfach und direkt auf alle benötigten Funktionen zugreifen.

Im Folgenden werden die GUIs des MasterClient und des ListenerClient beschrieben.

### MasterClient

Für den Vortragenden ergaben sich folgende zusammenhängende Interaktionsstränge:

1. Login und Authentifizierung
2. Vorbereiten, Steuern und beenden der Präsentation, sowie Beantworten von Fragen während der Präsentation
3. Einstellungen vornehmen



Entsprechend wurde ein Format mit 3 Tabs als passendes Konzept ausgewählt. Folgend werden die Schritte beschrieben:

### ***Login und Authentifizierung***


#### Anforderungen

Der Vortragenden muss sich mit dem Server per IP/Port Kombination anmelden können. Weiterhin muss er zur Authentifizierung ein Passwort eingeben.

---

<sup>10</sup> Autor: Jan Zimmer

## Umsetzung

IP-Adresse:	192.168.1.210
CMD-Port:	1337
Data-Port:	1338
Passwort:	..... 
<div>Login</div>	

Der Nutzer erhält die Möglichkeit per Textfeld IP und Ports anzugeben. Auch das Passwort wird per Textfeld eingegeben, zu beachten ist, dass dies per default verschleiert passiert. Über den Zentralen Knopf initiiert der Benutzer den Login.

Sobald der Loginstatus geändert wurde wird der Anwender per sogenanntem Toast, einer kurz eingeblendeten Nachricht, informiert.

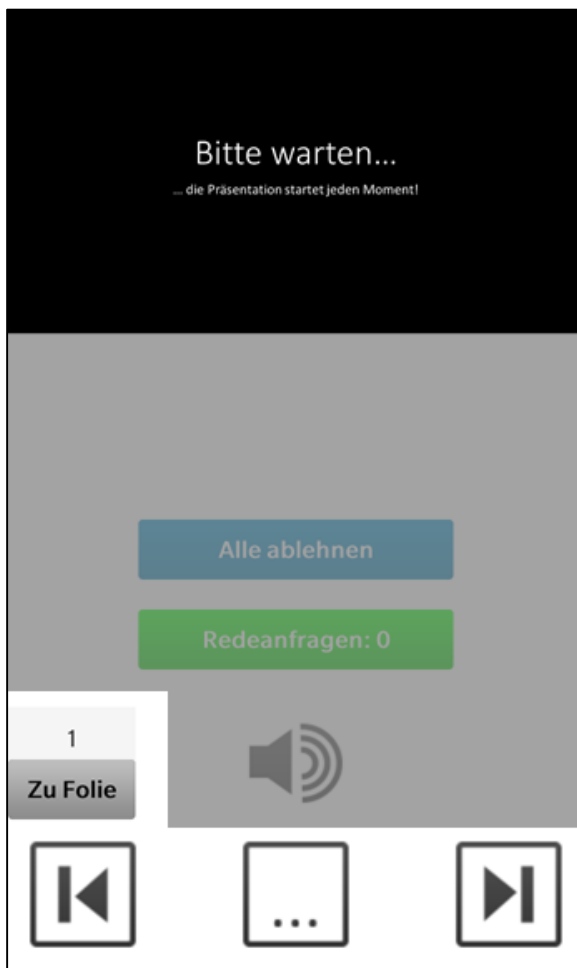
War der Login erfolgreich, so vibriert das Gerät zur Signalisierung kurz und wechselt dann automatisch in den Präsentationstab.

### ***Vorbereiten, Steuern und beenden der Präsentation, ...***

#### Anforderungen

Der Vortragende soll möglichst einfach die Präsentation auswählen können. Der Vortragende soll intuitiv durch die Folien navigieren.

## Umsetzung



Das obere Drittel des Tabs wird von der Präsentation ausgefüllt. Diese wird auch bei Interaktion des Nutzers so nie von Fingern verdeckt.

Am unteren Rand, und somit am leichtesten erreichbar sind die Navigationsknöpfe mit entsprechenden Piktogrammen platziert. Zur direkten Navigation wird ein Textfeld verwendet, das nur numerische Eingaben erlaubt. Wird eine Folie ausgewählt, die nicht in der Präsentation enthalten ist oder man erreicht das Ende der Präsentation, so wird dies per Toast signalisiert.

Der mittlere Knopf öffnet zu Beginn einen Dialog, der die Auswahl der Präsentation zulässt. Wird dieser geschlossen, wird die Präsentation automatisch geladen und an den Server versendet. Anschließend wird selbiger Knopf zu einem Stop-Button, der die Präsentation stoppt.

Gestartet wird die Präsentation, indem einmalig ein Folienwechsel ausgeführt wird. So wird auch automatisch die Audioaufnahme gestartet. Dies ist so gewählt, um

unnötige Navigationselemente zu sparen. Die laufende Aufnahme wird dabei mit einer blinkenden roten LED an der Frontseite signalisiert.

### ***... sowie Beantworten von Fragen während der Präsentation***

#### Anforderung

Die Anforderungen wurden bereits in Sektion 4.2 unter „Redeanfragen“ formuliert.



#### Umsetzung

Die Redeanfragesektion wird mittig oberhalb der Navigationssektion platziert. Sie hat so einen prominenten Platz, der leicht erreichbar und dennoch direkt im Blickfeld ist um etwaige Anfragen anzuzeigen.

Der Grüne Knopf zeigt gleichzeitig die Anzahl der Anfragen an und dient der Beantwortung dieser. Der blaue Knopf lehnte alle Anfragen auf einen Schlag ab.

Der Lautsprecher Knopf signalisiert, ob Anfragen angenommen oder direkt abgewiesen werden. Betätigt man ihn, so ändert sich auch sein Piktogramm entsprechend des aktuellen Status.

Zur Signalisierung wurde sowohl optisches als auch haptisches Feedback umgesetzt. Der Nutzer wird bei jeder Anfrage mit einem kurzen Vibrieren auf diese Hingewiesen. Gleichzeitig pulsiert der grüne Knopf nun Rot.





Die Antwort des Zuhörers wird dem Vortragenden per Toast signalisiert. Kommt eine Fragediskussion zustande, so werden alle Knöpfe der Sektion ausgeblendet und ein neuer Knopf, der die Anfrage schließlich beendet wird eingeblendet.



### ***Einstellungen vornehmen***

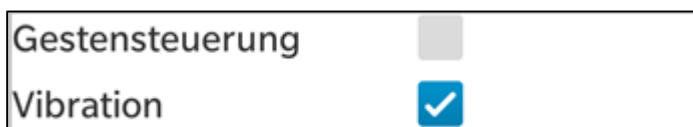
#### Anforderung

Der Nutzer soll die Gestensteuerung ein- und ausschalten können.

Neue Anforderungen: Da während der Nutzung der Gestensteuerung davon auszugehen ist, dass das Gerät auf einen Untergrund gelegt wird und gleichzeitig die Audioaufnahme läuft, kommt es zu Störungen bei Vibration des Gerätes.

Deshalb: Der Nutzer soll die Vibration ein- und ausschalten können. Bei Aktivierung der Gestensteuerung soll die Vibration automatisch deaktiviert werden.

#### Umsetzung



Als letzter Tab wird eine Liste von Einstellungen bereitgestellt. Diese sind als einfache Checkboxen implementiert. Das automatische Ausschalten der Vibration ist in QML realisiert.

### **ListenerClient**

Für den Vortragenden ergaben sich Folgende zusammenhängende Interaktionsstränge:

1. Login
2. Ansehen der Präsentation, sowie Redeanfragen stellen

Entsprechend wurden zwei Tabs erstellt, die im Folgenden beschrieben werden:

## **Login**

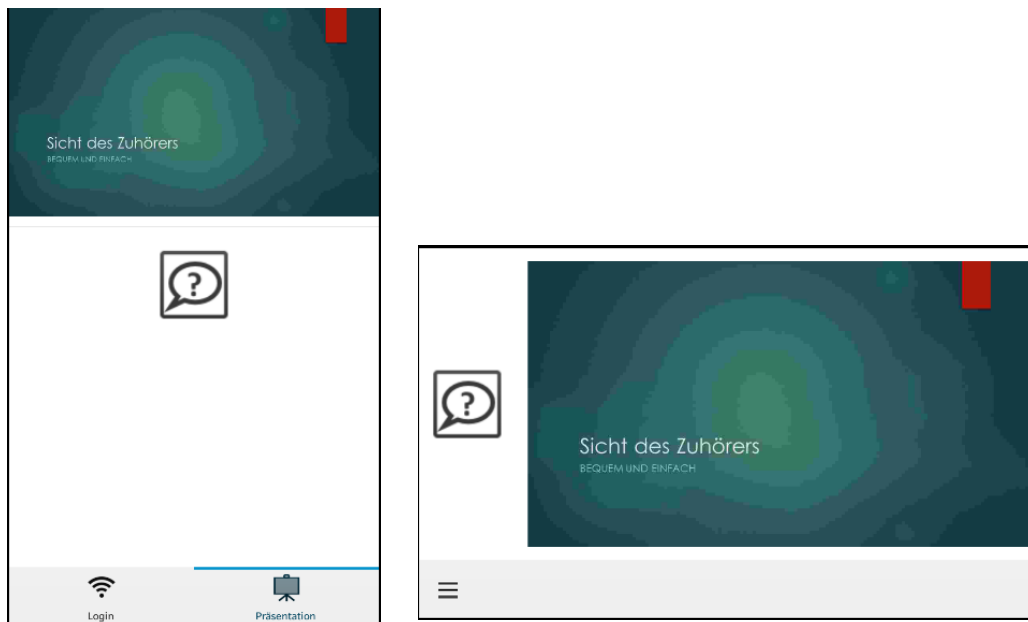
Der Logintab ist identisch zu dem des Masterclients, mit der Ausnahme, dass kein Passwort entgegengenommen wird. Auch das restliche Verhalten ist identisch.

## **Ansehen der Präsentation ...**

### Anforderung

Der Zuhörer muss komfortabel der Präsentation folgen können.

### Umsetzung



Dem Nutzer wird die Präsentation sowohl im Hoch- als auch im Querformat möglichst prominent präsentiert.

## **... sowie Redeanfragen stellen**

### Anforderung

Der Nutzer soll möglichst intuitiv Fragen stellen können.

### Umsetzung

Es wird ein Knopf mit Piktogramm zentral platziert, der die Anfrage auslöst. Solange die Anfrage in Bearbeitung ist, wird ein Aktivitätsindikator eingeblendet und der Knopf wird ausgeblendet.



Wird die Frage vom Vortragenden akzeptiert erscheint ein Dialog, der die Aktualität der Frage abfragt und es wird entsprechend der Nutzereingabe weiterverfahren.

Redeanfrage	
Redeanfrage noch aktuell?	
Nein	Ja

### Kritischer Kommentar

Mangels Kenntnis, wie man QML Elemente gemeinsam in zwei Applikationen verwendet, wurden viele Teile doppelt implementiert. Dies sollte normalerweise möglichst in einer gemeinsamen Basis getan werden.

Weiterhin wurde auf Internationalisierung kein Wert gelegt. In Zukunft sollte dies angestrebt werden.

## 4.6 Audio<sup>11</sup>

### Funktionalität und Anforderungen

Eine der grundlegenden Funktionen der Software ist die Aufnahme des Sprachsignals während des Vortrags. Dazu soll das interne Mikrofon des Telefons benutzt werden. Außerdem soll die LED während der Aufnahme rot blinken. (In einigen Ländern ist es gesetzlich vorgeschrieben, eine Ton- oder Bildaufnahme optisch durch ein rotes Blinken zu signalisieren.)

Um die Möglichkeit zur eindeutigen Unterscheidung zwischen den Aufnahmen zu gewährleisten, was für die (evtl. automatische) nachträgliche Bearbeitung zwingend erforderlich ist, wurde entschieden, diese mit dem Startpunkt (Stunde/Minute/Sekunde) der Aufnahme zu benennen (Dateiname).

---

<sup>11</sup> Autor: Arno Däuper

Es stellte sich nach kurzer Recherche heraus, dass das Framework bereits die grundlegenden Funktionen zur Audioaufnahme und auch zur Ansteuerung der LED bereitstellt.

## Struktur

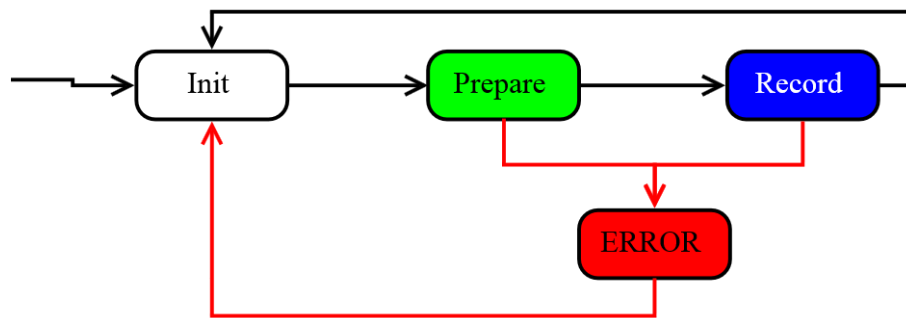
Die Implementierung umfasst eine Klasse (*Emaudiorecorder*), welche mithilfe von nur zwei Befehlen (*record* und *stop*) die vollständige Steuerung des Aufnahmevorgangs übernimmt. Es existieren keine Vererbungen oder Unterklassen.

## Details zur Implementierung

Die Implementierung erfolgte in der Sprache C++. Es werden die Klassen *bb::multimedia::Audiorecorder* sowie, zur Konfiguration, *bb::multimedia::AudioChannelConfiguration* genutzt. Um Zugriff auf das Mikrofon gewährleistet zu bekommen, ist es notwendig, die entsprechenden Zugriffsrechte in der *bar\_descriptor.xml* ('record\_audio') zu setzen. Zur Benennung des Dateinamens nach der Uhrzeit bietet das Framework einen Zugriff auf die Systemzeit über die Klasse *bb::sys::InvokeDateTime*.

Mit dem Telefon sind verschiedene Audiokonfigurationen möglich, so enthält es ein zweites Mikrofon, welches üblicherweise zur Störschallunterdrückung genutzt wird. Zur Vereinfachung wird in der Anwendung ausschließlich das Hauptmikrofon genutzt und somit ein monophones (einkanaliges) Signal aufgenommen.

Der *Emaudiorecorder* lässt sich in Form einer klassischen State-Machine beschreiben: Zu Beginn wird der *AudioRecorder* initialisiert und ein default-Dateiname vergeben. Bei Übergang in den *prepare-state* wird die Systemzeit ermittelt, in einen String überführt und an den *AudioRecorder* als Dateiname übergeben. Erst danach kann die Funktion *prepare* des Audiorecorders aufgerufen werden, welche diesen in den *armed-State* bringt. Dann wird die Aufnahme gestartet. Die Aufnahme wird schließlich angehalten, womit wieder der Ausgangs-State erreicht wird. In jedem Schritt ist es möglich, dass ein Fehler auftritt, weswegen die Rückgabewerte aus den Methoden der Klasse *AudioRecorder* ausgewertet werden sollten.



Die LED des Telefons, positioniert an der oberen rechten Ecke des Displays, wird über die Klasse *bb::device::Led* gesteuert und die Farbe über *bb::device::LedColor*. Um die Funktionsweise unabhängig von der Applikation zu überprüfen, wurde zunächst die Methode *Emaudiorecorder::LED\_TEST()* implementiert und danach der gewonnene Code in den Methoden *record* und *stop* verwendet.

Beide Funktionen – die Audioaufnahme und die LED – sind Threads, welche aber bereits vom Framework behandelt werden. Die LED blinkt ca. jede Sekunde, während der Audiothread mindestens mit der Samplerate von 48 kHz laufen muss. Dazu kommt eine Datenkompression, abhängig vom gewählten Format.

## 4.7 Bildschirmausgabe über HDMI<sup>12</sup>

### Funktionalität und Anforderungen

Das Telefon ist mit einem HDMI-Port ausgestattet, welcher regulär das Monitorbild des Telefons gespiegelt ausgibt. Während der Präsentation soll jedoch ein anderes Bild, nämlich die jeweilige Folie, ausgegeben werden, während sich auf dem Display des Vortragenden die Steuerelemente befinden.

### Struktur

Für die HDMI-Ausgabe existierte eine Library im Blackboard. Diese wurde genutzt, um – ähnlich wie bei der Audioaufnahme – eine sehr einfache API zu erstellen, die lediglich die Funktionen *show\_slide* und *show\_last\_slide* umfasst.

### Details zur Implementierung

Die Implementierung erfolgte ebenfalls in der Sprache C++. Im Konstruktor der Klasse wird die Bildschirmauflösung übergeben, so dass ein spontaner Wechsel des Ausgabegeräts nicht möglich ist. Es wurde ebenfalls auf eine Fehlerbehandlung (beispielsweise Abziehen des HDMI-Kabels) aufgrund der kurzen Entwicklungszeit verzichtet.

Zunächst bestand die Überlegung, die jeweils nächste Folie bereits in einer Art Cache vorzuladen, um den Übergang zu beschleunigen. Die nächste angezeigte Folie ist üblicherweise das nächste Bild, Sprünge zu einer beliebigen Folie könnten so nicht beschleunigt werden. In der Praxis stellte sich jedoch heraus, dass diese Maßnahme nicht notwendig ist, daher wurde der entsprechende Quelltext auskommentiert.

---

<sup>12</sup> Autor: Arno Däuper

## 4.8 Gestensteuerung<sup>13</sup>

### Funktionalität und Anforderungen

Diese Softwarekomponente ist dafür zuständig, dass es möglich ist, die laufende Präsentation mit Hilfe von Gesten zu steuern. Diese Art der Steuerung soll dem Anwender parallel zur klassischen Steuerung über die grafische Benutzeroberfläche zur Verfügung stehen. Dabei geht es nicht um Berührungsgesten, wie sie beim Blackberry bereits vielfältig eingesetzt werden, sondern um die Erkennung von Handbewegungen, die im Sichtfeld des Kamerasuchers vollführt werden.

Da dynamische Objekterkennung relativ schwierig zu realisieren ist, haben wir uns für folgende Einschränkungen entschieden: Via Gestensteuerung kann der Anwender nur zwei Eingaben tätigen – entweder eine Folie nach vorne oder eine Folie zurück schalten. Für die erste Funktion bewegt er seine Hand (oder ein anderes ausreichend dunkles Objekt) von links nach rechts über das Blackberry und für die zweite Funktion in die entgegengesetzte Richtung. Damit die Gestensteuerung zuverlässig funktioniert, ist es erforderlich, dass das Blackberry mit der Vorderseite nach oben stationär positioniert wird und der Hintergrund – also im Regelfall die Decke des Raumes – einheitlich hell ist. Für bestmögliche Ergebnisse sollten die Gesten zudem relativ langsam und über die gesamte Breite des Blackberry ausgeführt werden.

Um zu verhindern, dass Steuerungsbefehle ausgeführt werden, die nicht intendiert sind, ist die Komponente grundsätzlich so ausgelegt, dass im Zweifelsfall eine Geste eher nicht erkannt wird, als dass fälschlicherweise die angezeigte Folie gewechselt wird. Trotzdem ist es nicht auszuschließen, dass beispielsweise Eingaben über die grafische Benutzeroberfläche dazu führen, dass „falsche“ Gesten identifiziert werden, wenn sich die Hand des Anwenders dabei über den Kamerasucher bewegt. Daher lässt sich das Feature während des Betriebs aktivieren oder deaktivieren, um dem Anwender die freie Wahl zu lassen, wie er die Applikation bedient.

Aus Entwicklersicht ergeben sich aus der beschriebenen gewünschten Funktionalität folgende Anforderungen:

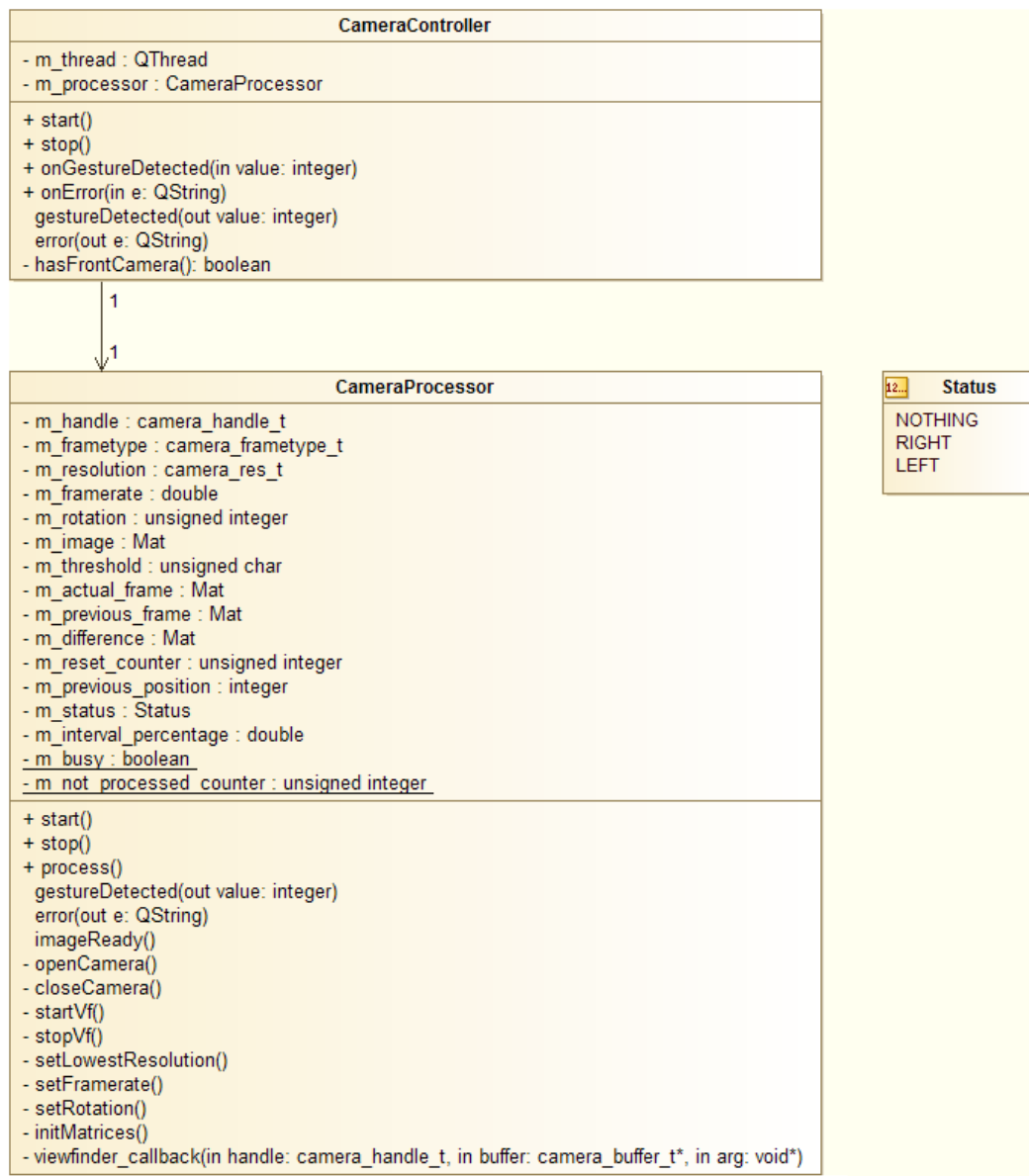
---

<sup>13</sup> Autor: Inga Quatuor

- Die Komponente nutzt die Frontkamera des Device.
- Die Gestensteuerung lässt sich aktivieren und deaktivieren.
- Die Gestensteuerung läuft in einem separaten Thread, um die restliche Funktionalität der Applikation nicht zu beeinträchtigen.
- Die Komponente verwaltet die Kameraressource und sorgt insbesondere dafür, dass der Zugriff korrekt beendet wird.
- Die Gestenerkennung ist so zu gestalten, dass die Bildauswertung in Echtzeit erfolgen kann und die Anzahl von *false positives* minimiert wird.
- Zusätzlich war geplant, die folgenden Aspekte zu berücksichtigen:
- Die Komponente soll nicht auf das Z10 spezialisiert sein, sondern auch andere Blackberrys unterstützen. Von daher sind sämtliche Kameraeinstellungen vor dem Einsatz zu überprüfen, ob sie vom Device unterstützt werden.
- Der Einsatz der Komponente durch die Hauptanwendung soll möglichst unkompliziert sein, so dass nach außen ein Interface angeboten wird und die Fachlogik abgeschirmt implementiert wird.
- Nur bei erfolgreich erkannter Geste oder im Fehlerfall wird die MasterClient-Anwendung informiert.



## Struktur



Die Komponente Gestensteuerung besteht aus zwei Klassen: der Interface-Klasse *CameraController*, die von der MasterClient-Anwendung genutzt wird und die notwendigen Funktionen zum Starten und Stoppen des Features beinhaltet und der Klasse *CameraProcessor*, welche die eigentliche Kamerasteuerung und Bildauswertung durchführt.

Um die Komponente einzusetzen, wird vom MasterClient beim Start der Applikation ein Objekt der Klasse *CameraController* erzeugt und die Signale dieser Klasse – *gestureDetected(int)* und *error(QString)* – werden mit passenden Slots verknüpft. Um die Gestensteuerung zu aktivieren, muss der Slot *start()* aufgerufen werden, und um sie zu deaktivieren, der Slot *stop()*.

Wenn *start()* aufgerufen wird, erzeugt die *CameraController*-Instanz ein Objekt der Klasse *CameraProcessor*. *CameraProcessor* löst dieselben Signale aus wie *CameraController*, so dass die Aufgabe des Controllers nur darin besteht, die Signale mit Hilfe entsprechender Slots an den MasterClient weiterzuleiten. Nachdem diese Signal-Slot-Verknüpfung erfolgt ist, verschiebt der Controller das Processor-Objekt in einen separaten Thread und ruft dessen *start()*-Funktion auf. Daraufhin aktiviert der *CameraProcessor* die Frontkamera des Blackberry, nimmt die passenden Einstellungen bzgl. Bildformat, Auflösung und Bildwiederholungsrate vor und startet den Kamerasucher. Sobald ein Einzelbild verfügbar ist, wird die Callback-Funktion *viewfinder\_callback()* ausgeführt und von dort die Bildauswertung an die Funktion *process()* abgegeben. Für die Bildanalyse wird die Open-Source-Bibliothek *openCV*<sup>14</sup> verwendet. Falls nach mehreren Einzelbildern eine Geste erkannt wurde, wird das Signal *gestureDetected(int)* ausgelöst, wobei der Parameter angibt, ob es sich um eine Geste nach links (-1) oder eine Geste nach rechts (+1) handelt.

### Details zur Implementierung

An dieser Stelle sollen drei Punkte genauer betrachtet werden. Erstens die Funktion *startVf()* in der Klasse *CameraProcessor*, in der die Kameraeinstellungen vorgenommen werden und bei deren Implementierung leider einige Schwierigkeiten aufgetreten waren. Zweitens der genaue Ablauf von *viewfinder\_callback()* und der anschließende Aufruf von *process()* insbesondere in Hinblick auf Nebenläufigkeit. Drittens die eigentliche Bildauswertung und Gestenerkennung innerhalb der *process()*-Funktion.

Zur Steuerung der Kamereinheiten werden von Blackberry zwei unterschiedliche APIs zur Verfügung gestellt.<sup>15</sup> Einerseits die *Cascades API*, die mit Hilfe von QML- oder C++-Code verwendet werden kann und Zugriff auf gebräuchlichen Kamerafunktionen liefert und einfach einzusetzen ist, aber auf einem recht hohen Abstraktionslevel arbeitet. Andererseits die *C API*, die auf einem niedrigeren Level ansetzt und einen größeren Funktionsumfang bereitstellt, aber etwas umständlicher zu implementieren ist. Da die

---

<sup>14</sup> OpenCV – Open Source Computer Vision. URL: [opencv.org](http://opencv.org) (zuletzt aufgerufen am 14.08.2015)

<sup>15</sup> Vgl. „The Camera APIs“ auf der Blackberry-Developer-Website. URL: [http://developer.blackberry.com/native/documentation/graphics\\_multimedia/camera/the\\_api.htm](http://developer.blackberry.com/native/documentation/graphics_multimedia/camera/the_api.htm) (zuletzt aufgerufen am 14.08.2015).

Einstellung der Bildwiederholungsrate für die Komponente geplant war und diese Funktion nur von der *C API* bereitgestellt wird, war schnell klar, dass diese API verwendet werden muss.

Da die Auswertung der Einzelbilder, die von der Kamera geliefert werden, in Echtzeit gemäß der Bildwiederholungsrate erfolgen muss, war es wichtig, die Bildinformationen möglichst gering zu halten. Daher sollte die Auflösung möglichst niedrig gewählt werden und das Bild in Graustufen aufgenommen werden. Da das Bild nur hinter den Kulissen verarbeitet wird, ist das Erzeugen eines Viewfinder-Windows nicht erforderlich. Die Framerate sollte zunächst niedrig gewählt werden (ca. 16 Bilder pro Sekunde) und kann, wenn die Performance unter dieser Bedingung zufriedenstellend ist, erhöht werden. Insgesamt war geplant, die folgenden Einstellungen für den Viewfinder innerhalb der *startVf()*-Funktion vorzunehmen: Ein Bildformat mit Graustufen, die kleinstmögliche Auflösung, die gewünschte Bildwiederholungsrate, die Rotation und das Deaktivieren der automatischen Fenstererzeugung. Bevor eine Einstellung gesetzt wird, soll überprüft werden, ob die gewünschte Einstellung unterstützt wird bzw. im Fall der Auflösung soll von allen unterstützten Auflösungen die niedrigste ausgewählt werden.

Das erste Problem an dieser Stelle trat bereits beim Kompilieren auf: Die Funktion *camera\_get\_supported\_vf\_frametypes()* aus der *C API* ist zwar im Header deklariert, aber nicht implementiert, so dass hierauf verzichtet werden muss. Der Versuch, das Bildformat auf GRAY8 zu setzen, scheiterte und erst durch Ausprobieren ließ sich feststellen, dass vom Blackberry Z10 wohl nur NV12 als Graustufen-Format unterstützt wird.

Das nächste Problem trat beim Setzen der Bildrotation auf. Hier wurde fälschlich von der Annahme ausgegangen, dass eine Rotation von 0 – also keine Rotation – mit Sicherheit unterstützt wird, so dass dieser Wert gesetzt wurde, wenn die gewünschte Rotation in *setRotation()* nicht gefunden wird. Tatsächlich unterstützt das Z10 jedoch ausschließlich eine Rotation um 90 Grad.

Nachdem diese Fehler ausfindig gemacht worden waren – was aufgrund der unspezifischen Fehlercodes viel Zeit in Anspruch genommen hat – ließ sich die Komponente fehlerfrei kompilieren. Zur Laufzeit ließ sich der Viewfinder aber trotzdem nicht starten. Nach zahllosen Versuchen mit auskommentierten Codeabschnitten stellte

sich heraus, dass von der Funktion *camera\_get\_supported\_vf\_resolutions()* aus der *C API* zwar 12 Auflösungen zurückgeliefert werden, aber nur drei hiervon tatsächlich funktionieren. Das heißt, es lassen sich alle 12 Auflösungen ohne Fehler setzen, aber nur bei dreien lässt sich der Kamerasucher anschließend starten. Von daher ist die Hilfsfunktion *setLowestResolution()* in der finalen Implementierung zwar noch enthalten, aber nach ihrer Ausführung wird die Auflösung (*m\_resolution*) von Hand auf 288 x 512 geändert.

Wie bereits erwähnt, wird die Klasse *CameraProcessor* in einem eigenen Thread ausgeführt, damit der Rest der Applikation nicht gestört wird. Wenn nun ein Einzelbild verfügbar ist, wird automatisch die Funktion *viewfinder\_callback()* aufgerufen, die wiederum in einem eigenen Thread ausgeführt wird und zudem eine statische Funktion ist, was bedeutet, dass sie unabhängig von der erzeugten *CameraProcessor*-Instanz arbeitet. Da für die Bildauswertung der Zugriff auf *Member Variables* von *CameraProcessor* jedoch erforderlich ist, musste eine Lösung gefunden werden, welche die Bildanalyse wieder zurück an das Processor-Objekt gibt. Dafür wird einerseits als letztes Argument an die Callback-Methode ein Zeiger auf das Processor-Objekt übergeben, so dass über diesen Umweg auf Variablen zugegriffen werden kann. Andererseits wird im Konstruktor des *CameraProcessor* ein Signal *imageReady()* mit dem Slot *process()* verknüpft und zwar in Form einer *Queued Connection*. Das bedeutet, dass der Slot nicht wie üblich im Thread des Senders ausgeführt wird, sondern im Thread des Empfängers.

Wenn *viewfinder\_callback()* nun aufgerufen wird, überprüft diese Funktion zunächst, ob noch eine Bildverarbeitung im Gang ist, was über die boolsche Klassenvariable *m\_busy* angezeigt wird. Falls nicht, wird das gebufferte Einzelbild in der Matrix *m\_image* gespeichert und das Signal *imageReady()* ausgelöst. Daraufhin übernimmt die *process()*-Funktion des Processor-Objekts (im anderen Thread) und verarbeitet diese Matrix weiter.

In der Funktion *process()* werden nun folgende Schritte durchgeführt, um zu entscheiden, ob eine Geste vorliegt oder nicht:

1. Das Bild wird rotiert, so dass die Ausrichtung dem im Hochformat liegenden Blackberry entspricht.

2. Das Bild wird auf die mittlere Pixel-Zeile reduziert, da eine horizontale Objektbewegung anhand einer einzelnen Zeile erkennbar ist.
3. Diese Zeile wird mit der vorherigen mittleren Zeile verglichen:  $m\_difference = m\_previous\_frame - m\_actual\_frame$ . Da die einzelnen Pixel Werte zwischen 0 (schwarz) und 255 (weiß) annehmen, ist dieses Differenzbild genau an den Stellen größer 0, an denen das neue Bild dunkler ist als das alte.
4. Das Differenzbild wird nun weiter reduziert, in dem alle Pixel, deren Wert größer ist als  $m\_threshold$ , auf 1 gesetzt werden und alle anderen Pixel auf 0. Durch den Threshold ist garantiert, dass nur signifikante Änderungen berücksichtigt werden. Sollten Bewegungen fälschlicherweise nicht erkannt werden, muss dieser Wert verringert werden. Werden nicht intendierte Bewegungen erkannt, muss er erhöht werden.
5. Falls  $m\_difference$  an mindestens einer Position ungleich 0 ist, gab es Veränderungen zum vorherigen Frame und es muss untersucht werden, ob eine Geste anfängt, weitergeführt oder vollendet wird. Das Verhalten der Funktion hängt hierbei davon ab, in welchem Status ( $m\_status$ ) sich das Processor-Objekt befinden:
  - Falls zuvor keine potentielle Geste erkannt wurde ( $m\_status = NOTHING$ ), muss untersucht werden, ob die Veränderung ganz links oder ganz rechts<sup>16</sup> im Bild stattgefunden hat. Falls ja, liegt hier ein Indikator für eine beginnende Geste vor und der Status wird entsprechend geändert.
  - Falls eine mögliche Geste nach rechts erkannt wurde ( $m\_status = RIGHT$ ), muss untersucht werden, ob sich diese Geste fortsetzt, also die neue Veränderung rechts neben der alten Veränderung positioniert ist. Falls nicht, wird die Geste verworfen und der Status wechselt zu NOTHING. Außerdem muss überprüft werden, ob die Geste abgeschlossen ist, das heißt, ob die neue Veränderung am rechten Rand des Bilds erfolgt ist. Falls ja, wird das `gestureDetected()`-Signal ausgelöst.

---

<sup>16</sup> Das Intervall, das als „ganz links“ bzw. „ganz rechts“ zählt, ergibt sich aus  $m\_interval\_percentage$ , dem prozentualen Anteil der gesamten Bildbreite, der ein Intervall bildet. Derselbe Wert wird verwendet, um den Bereich zu bestimmen, in dem überprüft wird, ob eine Geste von einem Frame zum nächsten fortgeführt wird.

- Falls eine mögliche Geste nach links erkannt wurde (`m_status = LEFT`), erfolgt die Verarbeitung analog zum vorherigen Punkt, nur in umgekehrter Richtung.

## 5 Systemintegration<sup>17</sup>

Die Entwicklung wurde mit dem Versionsverwaltungstool git und der Blackberry-IDE Momentics durchgeführt. Insbesondere durch die Momentics-IDE gab es jedoch verschiedene Probleme, die gelöst werden mussten.

Zunächst war es unser Plan, die Verwaltung innerhalb eines git-Repository auf mehrere Momentics-IDE-Projekte zu verteilen. Anfänglich war dazu die folgende Struktur angedacht:

- Basisverzeichnis
  - Clients
  - MasterAppl
  - ListenerAppl
  - Common
    - AudioLib
    - HdmiLib
    - GestureRecognitionLib
  - ServerAppl
  - Common
  - NetworkLib

Bei der anfänglichen Erstellung dieser Projekte traten zunächst Probleme auf, da die Momentics-IDE verschiedene Projektvorlagen für Library-Projekte bietet. Diese sind jedoch unzureichend dokumentiert, so dass erst nach einigen Startschwierigkeiten die richtige Vorlage genutzt wurde.

Bereits zu Beginn des Projektes hat sich die gewählte Struktur jedoch als nicht geeignet erwiesen. Der Hauptgrund dafür war, dass der Build-Prozess einzelner Anwendungen (wie zum Beispiel der Server-Applikation) zu umständlich und zeitraubend in der Bedienung wurde. Die Momentics-IDE stellt leider keine

---

<sup>17</sup> Autor: Sebastian Schwanewilms

automatisierte Funktion zur Verfügung, die den Build-Prozess verschiedener Library-Projekte vor dem Build einer Applikation durchführt.

Für ein längerfristiges Projekt wäre daher unter Umständen sinnvoll für den Build-Prozess eigene Skripte mittels CMake/QMake aufzusetzen. Auf Basis solcher Skripte könnte man einen automatischen Build-Prozess für jede einzelne Anwendung generieren. Dies würde zu einer deutlichen Vereinfachung in der Bedienung führen. Auch die Zeit, die ein Build-Prozess benötigt, würde deutlich sinken, da so ausschließlich einzelne Komponenten neu gebaut werden müssten.

Für dieses Projekt sind wir jedoch dazu übergegangen, die einzelnen Library-Projekte zu einer großen Library zusammenzufassen. Dies hatte zwar einen wesentlich einfacher zu bedienenden Build-Prozess zur Folge, dieser Prozess hatte jedoch eine relativ lange Laufzeit. Der Vorteil der einfacheren Bedienbarkeit überwiegt jedoch deutlich den Nachteil der Laufzeit.

Aus den genannten Überlegungen folgte abschließend die folgende Verzeichnis-/Projektstruktur:

- Basisverzeichnis
  - Client
    - ListenerClientAppl
    - MasterClientAppl
  - Common
    - ClientServerShareLib
  - ServerAppl

### **Build-Anleitung**

Es müssen drei Applikationen und eine Library gebaut werden. Da die Momentics-IDE leider keine Automatismen dafür anbietet muss dieses immer von Hand angestoßen werden. Es macht ebenfalls Sinn, die entsprechende Applikation und die Library vor einem neuen Build zu säubern.

Um einen Build durchzuführen, müssen zunächst die entsprechenden Projekte in einen Momentics-IDE-Workspace importiert werden. Dies kann am Einfachsten mittels der Import-Funktion der Momentics-IDE durchgeführt werden. Wenn man im Import-



Dialog die Variante „Existing Projects into Workspace“ wählt und im folgenden Dialog das Basisverzeichnis (siehe oben) als root-directory setzt, werden alle benötigten Projekte zur Auswahl angeboten. Es müssen die Projekte „ClientServerShareLib“, „ListenerClientAppl“, „MasterClientAppl“, „OpenCV“ und „ServerAppl“ importiert werden.

Es können Binaries für verschiedene Zielarchitekturen und Zwecke erzeugt werden. Es ist zu beachten, dass ein Binary für den Emulator für eine andere Architektur gebaut wird als für das Blackberry. Daher ist beim Säubern idealerweise die Anweisung „Clean All“ zu verwenden. Diese findet sich nach einem Rechtsklick im Project-Explorer auf ein Projekt in dem Unterpunkt „Build Configurations“. Dieses ist sowohl für das Library-Projekt (ClientServerShareLib) durchzuführen, wenn dort Änderungen vorgenommen wurden, und für das eigentliche Applikationsprojekt (ServerAppl, ListenerClientAppl oder MasterClientAppl).

Anschließend wird zunächst der Build-Prozess für die Library angestoßen. Dies geschieht über einen Rechtsklick auf das Projekt. Im Unterpunkt „Build Configurations“ findet sich dann der Befehl „Build All“. Dieser führt den Build-Prozess für die Library für alle Zielarchitekturen (Emulator/Blackberry) und Zwecke (mit oder ohne Debug-Symbole) durch. Im Anschluss ist ebenfalls der Build-Prozess für die entsprechende Applikation zu starten. Dies geschieht analog zum Build-Prozess des Library-Projektes.

## **Testing<sup>18</sup>**

Während des Projektes wurden die Implementierungen in zwei Phasen getestet. Während der Entwicklungsphase testete jeder Entwickler seine Teilbereiche autonom und manuell. Dazu wurden Teilweise Dummy-Applikationen eingerichtet, die genau diesem Zweck entsprachen. Anzumerken ist auch, dass die Gewählte Softwarearchitektur zuließ, einen schnell in Python entworfenen Server zum Testen der Client Applikation noch vor Fertigstellung des eigentlichen Server möglich machte und so die Systemintegration deutlich beschleunigte.

Die zweite Phase des Testings wurde während der schrittweisen Systemintegration vorgenommen. So wurden manuell alle Testfälle und möglichen Kombinationen

---

<sup>18</sup> Autor: Jan Zimmer

durchgespielt. Trat ein Fehler auf, so wurde dieser von den beteiligten Entwicklern eingegrenzt, beseitigt und das Testing wurde abermals auf den entsprechenden Testfall angewendet. Dieses iterative Verfahren ermöglichte die systematische Korrektur von Fehlern. Gleichzeitig konnte durch die gemeinschaftliche Bearbeitung des Fehlers der Quellcode an den kritischen Stellen vom jeweiligen Partner gegengelesen werden.

Anzumerken ist, dass explizit kein automatisiertes Testing durchgeführt wurde. Dies ist mit dem Knappen Zeitplan begründet.

Es wurde auch kein Speicherprofiling durchgeführt; allerdings wurde vorab eine Richtlinie festgelegt, die besagt, dass jede Komponente intern für das freigeben der verwendeten Ressourcen verantwortlich ist. So sollten zumindest zwischen den einzelnen Komponenten Speicherlecks vermieden werden.

## 6 Projektreview<sup>19</sup>

Während und nach der Arbeit am Projekt wird deutlich, welche der organisatorischen Aspekte funktioniert haben und welche im nächsten Projekt verbessert werden sollten. Dazu lässt sich festhalten, dass die Aufteilung nach technischem Aspekt absolut richtig war und die wöchentlichen Meetings, sowie ständige Kommunikation zum Erfolg des Projekts maßgeblich beigetragen haben.

Auch die Verwendung von Git zur Versionsverwaltung sollte beibehalten werden, da dies den Entwicklungsvorgang maßgeblich beschleunigt und wertvolle Erfahrungen mit dessen Umgang gesammelt werden konnten. In diesem Falle hat die Verwendung von Git jedoch eine nicht unerhebliche Verzögerung verursacht, da ein Teil der Entwickler keinerlei Erfahrungen im Umgang mit dem Werkzeug hatten und somit eine Einarbeitung notwendig war, bzw. auf die Benutzung verzichtet wurde. Dadurch bedingt konnte der Zeitplan nicht eingehalten werden sodass die Zusammenführung und das Testen unter massivem Zeitdruck stattfand.

Auch aufgrund von Zeitmangel wurden die Spezifikationen, die zu Beginn des Projekts verfasst worden waren, nicht ausreichend systematisch umgesetzt. Ein ausschlaggebender Grund dafür war die größtenteils unbekannte Entwicklungsumgebung mit ihren umfangreichen Möglichkeiten, sowie das mächtige Framework, dessen Funktionsumfang im Rahmen eines solchen Projekts kaum überschaubar ist.

---

<sup>19</sup> Autor: Arno Däuper

## 7 **Fazit**<sup>20</sup>

Bis zur Abschlusspräsentation der Applikation am 15. Juli 2015 wurden alle standardmäßig vorausgesetzten Bestandteile der App umgesetzt und in der Präsentation vorgestellt. Zusätzlich wurde die Entwicklung der von uns geplanten Zusatzfunktionen vor der Präsentation fertiggestellt und in die Applikation eingepflegt.

Abgesehen von der automatisierten Zusammenstellung von Audiomitschnitten und Folien nach Abschluss einer Präsentation, wurden alle Zusatzziele erfolgreich umgesetzt. Das beinhaltet die folgenden Erweiterungen zum Standardprojekt: Redeanfragen von Zuhörern, Audiomitschnitte des Vortragenden und Zuhörern und authentifizierte Kommunikation zwischen Masterclient und Server.

Rückblickend lässt sich sagen, dass die Kommunikation und die Arbeit innerhalb des Teams sehr gut und größtenteils problemlos zu einem erfolgreichen Ergebnis der Applikation führten. Hiermit möchten wir uns bei allen Teammitgliedern für die gute Kooperation und die geleistete Arbeit bedanken.

---

<sup>20</sup> Autor: Niklas Kröger

## 8 Anhang

### **Vollständige Liste der verwendeten Kommandos:**

#### CMD SET SLIDE "set slide"

Der Befehl SET\_SLIDE wird vom Master-Client verwendet um dem Server mitzuteilen, dass eine neue Folie innerhalb der Präsentation angezeigt werden soll. Der Server wird den gleichen Befehl verwenden, um die Listener-Clients zu informieren. Auch wird dieser Befehl an den Master-Client übermittelt. Dies gewährleistet ein synchrones umschalten der Folien auf allen Clients.

#### CMD STOP PRAESENTATION "stop praesentation"

Dieser Befehl wird verwendet, wenn der Master-Client die Präsentation beenden möchte. Er wird zunächst an den Server geschickt und von dort aus an die verschiedenen Listener-Clients übermittelt. Er wird vom Server ebenfalls, wenn die Netzwerkverbindung zum Master-Client unerwartet abreist. Es werden dadurch alle Listener-Clients über das Ende der Präsentation informiert.

#### DATA PRAESENTATION "deliver praesentation"

Der Befehl DATA\_PRAESENTATION wird vom Master-Client verwendet, um eine neue Präsentation an den Server zu übermitteln. Als Parameter enthält eine Nachricht mit diesem Befehl die verschiedenen Folien als Bilddateien. Die Bilddateien sind dabei Base64-kodiert. Der Server wird diesen Befehl ebenfalls verwenden, um die Präsentation an die Listener-Clients weiter zu leiten.

#### CMD LOGIN "login"

Listener-Clients verwenden diesen Befehl, um sich am Server anzumelden. Der Server erkennt hieran die Listener-Clients.

#### CMD LOGIN RESP "login resp"

Der Server versendet diesen Befehl, nach Erhalt eines LOGIN-Befehls. Er bestätigt dem Listener-Client die erfolgreiche Anmeldung.

#### CMD AUTH PHASE1 "auth phase1"

Master-Clients verwenden diesen Befehl, um sich am Server anzumelden. Der Server erkennt hieran einen Master-Client. Es wird die 1. Nonce mitgesendet.

#### CMD AUTH PHASE2 "auth phase2"

Der Server sendet dieses Kommando mit der 2. Nonce an den Master.

#### CMD AUTH PHASE3 "auth phase3"

In dieser Nachricht sendet der Master den „Beweis“, dass er im Besitz des Passwortes ist.

#### CMD AUTH PHASE4 "auth phase4"

Der Server antwortet, ob die Authentifizierung erfolgreich war.

#### CMD ACK RESPONSE "ack"

Wird als Acknowledge verwendet.

#### CMD RANF ASK "redeanfrage request"

Ein Zuhörer sendet dieses Kommando über den Server an den Master um eine Redeanfrage zu tätigen.

#### CMD RANF RESP "redeanfrage antwort"

Enthält die Antwort des Masters für entsprechende Anfrage.

#### CMD RANF RE RESP "redeanfrage finale antwort"

Enthält information, ob Redeanfrage noch aktuell ist.

#### CMD RANF FINISH "redeanfrage finish"

Beendet gegebenenfalls eine laufende Anfrage.

#### DATA AUDIO "deliver audio"

Dieser Befehl wird verwendet um Audioaufzeichnungen an den Server zu übermitteln. Es kann sich dabei um Aufzeichnungen von Redeanfragen durch

Listener-Clients handeln oder aber auch um eine Aufzeichnung der Gesamten Präsentation vom Master-Client.